The Pennsylvania State University
The Graduate School
College of Engineering

IMPROVING PERFORMANCE AND ENERGY EFFICIENCY FOR ON-DEVICE VISUAL COMPUTING

A Dissertation in
Computer Science and Engineering
by
Ziyu Ying

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2024

The dissertation of Ziyu Ying was reviewed and approved* by the following:

Mahmut T. Kandemir
Distinguished Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Mahanth Gowda
Wormley Family Early Career Assistant Professor of Computer Science and Engineering
Committee Member

Sharon Huang
Professor of Information Sciences and Technology
Committee Member

*Signatures are on file in the Graduate School.

Abstract

This is my abstract.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This is my acknowledgement.

Dedication

Chapter 1
Introduction

The current digital age is experiencing a rapid increase in visual content, which fundamentally reshapes the way we live and work. Visual content has become ubiquitous across various domains such as education, entertainment, sports, augmented/virtual reality (AR/VR), and more [10–16]. A survey by xx predicts that by 2025, video alone will account for 82% of consumer internet traffic [], which clearly demonstrates the growing prevalence of visual content.

In parallel to this explosion, edge devices such as smartphones, AR/VR headsets, and smart cameras, are becoming increasingly popular. These devices are steadily gaining computational abilities to handle visual computing tasks, which allows for real-time analytics, low-latency interactions, and personalized experiences by directly processing the visual data at the point of capture. According to a recent report from Apple, 6 of the top 10 freely downloaded apps on iPhone are visual-related, including Tiktok, Youtube, CapCut, Max, and others. This trend clearly indicates the undeniable shift towards visual computing on edge devices.

While the on-device visual computing promises significant benefits like reduced data transmission and enhanced privacy, it faces a fundamental challenge: the inherent resource limitations of edge devices. Despite their increasing capabilities in terms of processing power, memory, and bandwidth, these devices still struggle to handle complex visual data processing tasks, particularly when incorporating resource-intensive deep learning and computer vision algorithms. Moreover, the recent advancements of edge devices in capturing 3D visual data, such as point clouds, further amplify this challenge due to their substantial data volume and processing complexity. Therefore, optimizing on-device visual computing to enable faster, more energy-efficient processing and deliver higher-quality visual experiences is of paramount importance. This requires a holistic understanding of the entire visual computing pipeline on edge devices, encompassing

data acquisition, preprocessing, encoding, decoding, analytic/processing, and beyond.

While stages like capturing, preprocessing, encoding, and decoding for planar visual content have been extensively optimized using various algorithmic and hardware solutions [] and can run efficiently on edge devices (e.g., iPhones can easily support xx FPS for xx resolution), the analytics/processing stage, often reliant on complex DNN models for enhancing user interaction quality and accuracy with visual content, emerges as a critical bottleneck on edge devices. Prior efforts have tackled this issue from various perspectives, including DNN model compression and quantization (algorithm) [], automatic optimization and parallelization (compiler) [], and designing specialized hardware accelerators like FPGA, NPU, TPU, or ASIC []. However, these approaches often lack a system or application-specific perspective, such as *how to better utilize existing intellectual properties (IPs) on edge devices* or *how to reduce computation by considering and leveraging unique characteristics of the application or user behavior*. Instead, they are generally focused only on the DNN models, neglecting other aspects. While some efforts take the system- or application-level opportunities into consideration [], most of them target optimizing traditional or common computer vision applications, such as enabling efficient object detection, object recognition, and face recognition on edge devices []. However, less attention has been given to other critical yet time-consuming computer vision applications such as DNN-based high dynamic range (HDR) content generation on edge devices, which is crucial for providing a satisfactory user experience. According to our experiment, generating a single frame of HDR content in 1K resolution (xx*xx) using the SingleHDR model [] (a DNN model used for reconstructing HDR images from LDR inputs) on an edge GPU SoC (NVIDIA Jetson Orin Nano development board []) can take up to 4 seconds, highlighting the need for optimizing such tasks.

Furthermore, on-device processing for 3D visual data, such as point cloud data, which is vital for lots of applications like autonomous driving (enabling higher accuracy for better decision-making), AR/VR (providing more immersive user experience), and more, has received even less attention. This is because, due to high computational requirements and limited capturing capabilities of earlier edge devices, point cloud data was typically processed in the cloud or on desktop computers. However, with recent advancements, edge devices like the iPhone 13 Pro Max, are equipped with LiDAR cameras and capable of capturing point cloud data. This motivates exploring local processing for such data to avoid transmission latency and the cloud-related costs. However, unlike the planar visual content, which has ralatively matural hardware support (e.g., the codec hardware designed for encode/decode the planar visual data), there are no such specialized hardware

for 3D point cloud processing on edge devices. As a result, even the basic tasks like point cloud compression (PCC) become time-consuming (e.g., 4seconds/frame based on our profiling for the PCC on NVIDIA Jetson Xavier board []) , let alone DNN-based analytics, which are even more computationally intensive. Therefore, optimizing the processing of 3D visual data such as point clouds on edge devices is another crucial area that this dissertation will explore.

To summarize, the resources limitations of edge devices pose a significant challenge to unlocking the full potential of on-device visual computing, particularly for the complex tasks involving DNNs and 3D data. Existing optimization approaches often overlook a system-level or application-specific perspective, and 3D visual data processing on edge devices remains largely unexplored. This dissertation aims to address these gaps by taking a comprehensive investigation of the entire visual computing pipeline on edge devices, exploring system-level and application-specific optimization opportunities, and tailoring solutions both planar and 3D visual data. To this end, we adopt a three-pronged strategy – 1). systematically study the pipeline to identify the bottleneck stage(s) and the potential opportunities overlooked by the state-of-the-art on-device visual computing systems; 2). leverage the identified opportunities to design novel optimization schemes that address the specific bottleneck(s); 3). efficiently utilize the underlying hardware resources on edge devices to improve the performance and energy efficiency. We focus on four representative applications crucial for visual computing on edge devices, ranging from traditional 2D video to cutting-edge High Dynamic Range (HDR) visual content and 3D point clouds, encompassing the entire visual computing pipeline from compression to analytics, and leads to four significant contributions:

Contribution-I: Optimizing the DNN-based 2D video Analytics. We first target the DNN-assisted 2D video analytics pipeline on a typical mobile device and observe that the DNN inference is the most time-consuming stage due to the intensive computations and large memory footprints of the DNN models, as well as the high data volume of the video stream. Note that, unlike performing DNN inference on a single image, there exists rich information redundancy/similarity among consecutive video frames, therefore, providing us the potential opportunities for memorization and reuse. Our similarity studies on successive video frames show that, for the pixels at the corresponding positions of two adjacent frames, about 95% of their absolute differences are less than 3. Motivated by this, we investigate two simple, yet effective optimizations – reusing the computations at the frame-level (i.e., for the whole frame) and region-level (i.e., for the background regions) to boost the performance of the inference process and minimizing

the power/energy consumption. Further, as the "exact pixel match based similarity" is too strict and will limit the scope for optimization, in this proposal, we consider an alternate similarity metrics that utilize the motion vector to measure the similarity across frames and make the reuse decision. More importantly, the motion vectors can be obtained using the equipped codec hardware with minimal overhead, giving us a chance to capitalize on an existing fundamental component in the vision pipeline, instead of adding new hardware blindly.

Contribution-II: Efficient HDR Content Generation. (I will add this later.)

Contribution-III: Optimizing the point cloud Compression. Unlike the conventional planar videos, for which the processing pipeline on mobile/edge devices is relatively mature, there exist very few customized hardware/accelerators for volumetric video processing, leading to more bottlenecks that have to be solved. One such example is the video compression/encoding. While the codec hardware on current commercial edge/mobile devices is designated for compressing 2D videos, volumetric videos (e.g., point clouds) can hardly benefit from it. Our experiments on a typical edge SoC (i.e., Jetson AGX Xavier [17]) show that the point cloud compression consumes as high as $4.2s$ per frame and, hence, becomes the earliest bottleneck in the entire pipeline. The main reason for this inefficiency is the lack of parallelization in the state-of-the-art approaches, meaning that, all the points are processed sequentially. Towards this, we propose two complementary schemes, intra-frame compression and inter-frame compression with (significantly) increased parallelism to minimize the compression latency and energy consumption, without losing much quality or compression efficiency.

Contribution-IV: Efficient DNN Analytics for Point Clouds. While our third proposal addresses the inefficiencies of point cloud compression on edge devices, point cloud analysis is another bottleneck stage in the point cloud processing pipeline and is yet to be solved. Especially, the compute-intensive DNNs have been recently introduced into point cloud analytics to assist the accuracy-sensitive applications such as autonomous driving, making it even more challenging to efficiently analyze the point clouds. On the other hand, unlike the DNN analytics on 2D image data (which are structured), DNN on point clouds faces several challenges as the raw point cloud data are *irregular* and *unstructured*, which not only introduce the irregular memory access overhead, but also prevent the algorithm developers from utilizing the existing DNN accelerators designated for 2D data. Motivated by this, we propose a mechanism for using the *Morton Code* to *structurize* the raw point cloud data. We plan to implement our proposal on an NVIDIA AGX Xavier edge GPU board and evaluate our design with six different workloads.

This dissertation is organized as follows. Chapter xx.

Chapter 2
Background and Related Work

This section provides a brief background and related work about (i) optimizing the DNN inference for 2D videos; (ii) HDR content reconstruction; (iii) state-of-the-art compression techniques for point cloud data; (iv) optimization for the DNN-based point cloud analytics.

Chapter 3
Exploiting Frame Similarity for Efficient Inference on Edge Devices

3.1 Introduction

While video processing has become extremely popular on mobile devices, the next wave of emerging applications are likely to be those that analyze videos in *real-time*, to provide sophisticated intelligence. Such analytics are critical for virtual/augmented reality (in sports, gaming, scientific exploration), object recognition/detection for surveillance, commercial advertisement insertion/deletion, synopsis extraction, and video querying. To enable such analytics, current devices rely heavily on deep neural networks (DNNs). However, DNNs are quite compute-intensive and generate very large memory footprints [18]. Furthermore, running inference on video data on edge devices is even more expensive than standard image data because of the data volume and the power/energy constraints.

There has been recent research in optimizing video analytics on edge devices, targeting smartphones, autonomous driving cars, and VR/AR headsets. These optimizations have been attempted at different levels, including model compression (pruning [19, 20], quantization [**?**, 21–25]), compiler support [19, 26–29], runtime systems [1, 30–37], and hardware enhancements [38–42]. Let us now summarize the pros and cons of four state-of-the-art works and compare them with our work. DeepCache [1] has been recently proposed for trading layer-wise intermediate data memoization/caching for computation savings for all frames, but it ignores the frame-wise data reuse opportunities. From the hardware side, Euphrates [30] targets the frame-wise reuse, and customizes an accelerator for searching the region of interest. Different from these two prior works, Potluck [36] caches the input data (actually, the feature vectors (FVs) extracted from the input frames) and its inference result in memory, and directly returns the result for other frames with

similar FVs, with the costs of FV extraction (e.g., down-sampling [43]) penalty and potential accuracy and/or performance drop due to sampling failures. On the other hand, MCDNN [37] proposes a scheduler to dynamically choose the best suitable model from various models available to the system (e.g., standard model, medium-pruned variant, and heavily-pruned variant), and pushes the accuracy as high as possible within the energy budget. However, with limited energy budget on typical edge devices, the accuracy is far from sufficient for vision applications (quantitative results in Sec. 5.6).

Table 3.1: Qualitative summary of DNN vision optimization work in terms of accuracy, saving, adaptivity, hardware support, and decision making mechanism (DM). Note that a check-mark in the second and third columns means the corresponding scheme achieves high accuracy and energy/performance efficiency, respectively. A check-mark in the Custom HW column indicates that the corresponding approach does not need custom hardware, whereas a cross means it needs.

	Accuracy	Saving	Adaptivity	Custom HW	DM
MCDNN [37]	✗	✓	✓	✓	Resource Budget
Potluck [36]	✓	✗	✓	✓	Feature Vector
Euphrates [30]	✓	✓	✗	✗	Reuse Distance
DeepCache [1]	✓	✗	✗	✓	Reuse Distance
This Work	✓	✓	✓	✓	Motion Vector

Although the aforementioned schemes do add value to video processing, each comes with its own problems and costs. In comparing these prior approaches with ours, we consider five critical features of DNN-based video optimization, shown in Table 3.1: high accuracy, high performance improvement and energy savings, adaptation to various runtime conditions, the hardware enhancement needed, and generality of decision making logic for proper approximation. Specifically, MCDNN suffers from accuracy drops when the energy budget is limited or the number of models to choose from is small. Also, if the approximation opportunity incurs high overhead and/or the decision is made based on high-level features, the scope for performance/energy savings can be relatively low (as in Potluck [36] and DeepCache [1]). Additionally, sometimes sudden changes in subsequent frames (e.g., with a new object in the frame), require adaptive decisions, which cannot be achieved by Euphrates [30] and DeepCache [1]. Also, as opposed to Euphrates, one may prefer to use existing hardware, due to considerable development effort and associated cost with customization. Unlike the work presented in this paper, none of the four prior schemes mentioned in Table 3.1 performs well in all features listed in the table.

While one may hypothesize that a new software-hardware co-design may be a panacea for numerous applications, video processing at the edge is not necessarily in this category. Edge devices are often equipped with mature and robust video and vision pipelines,

including sophisticated hardware like the codecs, which the current video analytics pipeline is yet to fully exploit. Furthermore, the video stream itself has temporal continuity, giving us the opportunities of reuse and memoization. In special cases, like remote surveillance, most frames can even be identical, avoiding the need to process them. Consequently, the only regions of interest (RoIs) should be the change between successive frames. Furthermore, these changes between consecutive frames need not be explicitly calculated as they can be extracted from the hardware codecs directly, giving us a chance to capitalize on an existing fundamental component, instead of adding new hardware blindly.

Leveraging inter-frame similarity, intra-frame redundancies, and their RoIs have been investigated for different purposes, e.g., exploring pixel-similarity to compress video frames and optimize transfer bandwidth [44], analyzing inter-frame similarity to reduce the number of inferences [30], and memoizing the computational results of regions in a frame to avoid redundant computation [1]. However, the granularity of similarity explorations in these prior works is *static*, and either too coarse (e.g., skips the entire frame [30]) or too strict (e.g., still needs to compute despite very few changes). Hence, properly exploiting these concepts together in an "integrated" fashion for better optimizing neural inferencing is very much in its infancy that this paper proposes to address. Specifically, we try answering the following critical questions: *(i) what is the scope to explore both the pixel- and computational-similarities, i.e., frame-level, region-level or pixel-level?*, *(ii) can we exploit the temporal continuity of the video steam and safely skip the inference computation for similar frames?*, and if not, *(iii) can we just focus on the RoIs in the current frame and significantly reduce the computational requirements for its inferencing?*

Towards this, we propose and thoroughly evaluate a new approach trying to answer the aforementioned questions. Specifically, our approach combines inter-frame similarities, motion vectors (MVs), and the concept of regions of interest, to make online video analytics/inferences inherently faster, along with the runtime support for improving the video streaming pipeline. Our main **contributions** in this work include the following:

- First, we study the similarities between successive frames in videos at various levels. We identify *online-pruning* opportunities for inferences, which can also be exploited at *frame-level*, *region-level*, and *pixel-level*.
- We then propose a frame-level motion vector based scheme to leverage the frame-level similarity to opportunistically skip the inference computation by reusing the compute results memoized by the previous frame. In order to maintain high accuracy, we also propose an adaptive technique to dynamically adjust the reuse window size based on

the runtime statistics by comparing the MVs in consequent frames.

- Next, we propose a tile-level inference scheme to enable the region/tile reuse by identifying the critical regions, including the bounding boxes from the previous frame as well as the motion vector tiles when decoding the current frame. Since processing these critical regions is in general much lighter-weight than processing a whole frame due to smaller size and less computation, the proposed partial inference is able to minimize the unnecessary computation (on the background regions or unimportant regions), thereby speeding up the performance and reducing the energy consumption.
- Then, we implement and experimentally evaluate our proposal on a representative mobile device – the Pixel 3 Phone [45] – and collect detailed experimental results, using 6 different video streams. Our proposed frame-level reuse shows $\approx 53\%$ of the frames to be redundant and hence skips the inference, leading to only less than 1% accuracy loss. Combining the frame-level reuse with the partial inference gives 2.2× performance improvement on average, and up to 80% energy savings, while losing less than 2% accuracy.
- Finally, We compare our approach against four state-of-the-art works (DeepCache [1], Euphrates [30], Potluck [36], and MCDNN [37]). We outperform the MCDNN [37] and Euphrates [30] in terms of accuracy, and Potluck [36] and DeepCache [1] in terms of performance improvement.

3.2 Background and Related Work

In this section, we start our discussion by explaining a typical DNN execution on mobile devices for video analytics. We then go over potential optimization opportunities that have been explored by prior works to make DNN inference mobile-friendly. We focus mainly on YOLO models (e.g., YOLOv3 [46] and YOLOv4-tiny [47]) as our workload.

3.2.1 Video Inference on Mobile Platforms

The key difference between a video-based object detection application and other popular DNN inferencing applications, such as natural language processing (NLP), speech-to-text, etc., is that, the former interacts with video frames which are either captured from the camera or downloaded/streamed from internet and hence, has a strict latency requirement for performing inferencing within the frame deadline. As shown in the "HW" (Hardware) layer in Fig. 6.1, a typical mobile neural network video inference system has two major

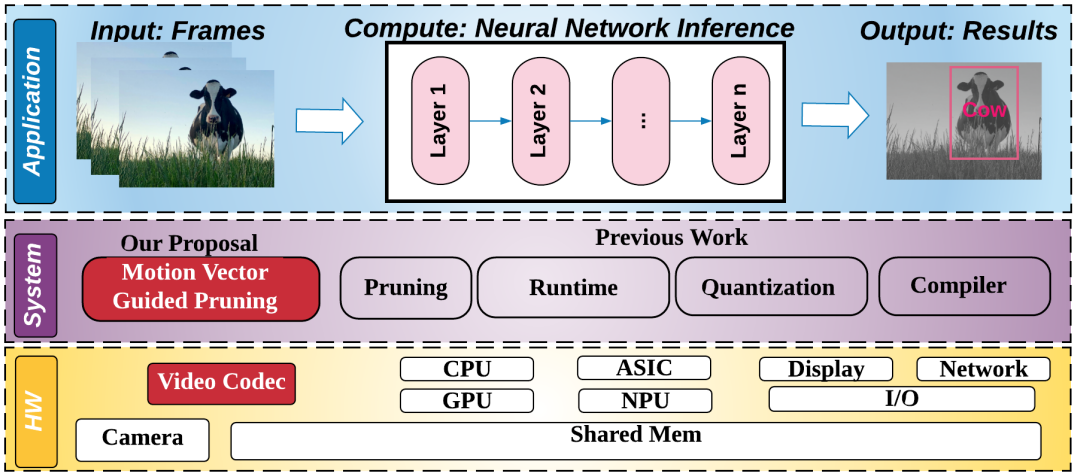


Figure 3.1: A DNN inference pipeline on an edge device with optimizations in the application, system, and hardware levels.

hardware components: (i) an SoC with a CPU/GPU/NPU for processing the intensive inference tasks, and an intermediate buffer in DRAM for storing frames captured by the camera as well as the intermediate data between layers of DNNs, and (ii) a video decoder communicating with the SoC ,typically via the memory bus. Running on top of the hardware, as shown in the "Application" layer in Fig. 6.1, the neural network video inference software pipeline can be summarized as follows:

Input: The raw video data is first pre-processed into pixels in the RGB/YUV domain by on-chip ISP, and then stored in memory (usually in the H.264/MPEG format). A hardware-based H.264/MPEG decoder decodes the compressed video bitstreams to obtain the original video frames. These decoded frames are then buffered in a memory buffer, waiting to be processed by the next stage – NN Inference.

NN Inference: In this stage, the neural network (NN) pipeline takes the decoded video data to perform the inference tasks with the available compute engines (e.g., CPU, GPU, NPU, or any dedicated ASICs). Numerous prior studies (e.g., see [48–50] and the references therein) have clearly shown that, regardless of the type of the compute hardware employed, the NN inferences are both compute and memory intensive. As a result, this stage is the main bottleneck in the NN applications.

Output: Following the inference stage, the resulting Feature-Maps (FMs) are used to generate the final tags/bounding-boxes and finally report to the application (e.g., a cow has been identified in the image with 95% confidence).

3.2.2 Prior Work

Prior work for enabling inferences on edge devices have focused on hardware as well as software optimizations, which can be further classified into model compression and pruning, and compiler and runtime support.

3.2.2.1 Hardware Optimizations

Traditionally, CPUs and GPUs have been recruited to execute DNN inference on mobile phones. Although DNN inference is highly structured and embarrassingly parallel, the limited resource budget of the mobile devices, alongside the required off-chip data movements, poses a significant challenge leading to higher latency. Several hardware-based approaches such as NPUs [51] or ASICs [52–54] have been proposed. These hardware approaches can significantly speedup the inference; however, only on a narrow set of specific applications and/or platforms.

3.2.2.2 Model Compression and Pruning

To make DNNs mobile friendly, there have been several works in compressing and pruning large models. While model compression [55–59] tries to compress the entire weight matrix while preserving accuracy, pruning goes over individual weights/kernels and drops the ones which contribute minimally towards the accuracy. Model compression is typically achieved by tensor decomposition or low rank compression, i.e., by representing the higher dimensional parameter matrix in a compressed low rank form. In contrast, pruning looks at the contribution of each individual weight. While some of the weights contribute more towards accuracy, some contribute less; and the ones contributing less are dropped to reduce the parameter size as well as compute and memory footprints. On the other hand, quantization, which has recently gained a lot of traction, trying to quantize the weights and activation to lower precision [52]. Combined with specialized hardware support, quantization can reduce the amount of computation, memory footprint, and energy consumption. Note that these compression and pruning approaches can be performed at compile time, and they typically need fine-tuning to retain accuracy.

3.2.2.3 System Support for Exploiting Pixel and Computation Similarities

State-of-the-art proposals such as DeepCache [1], and Euphrates [30] have explored the temporal similarity at runtime for DNN inference on video streams. Therefore, in Sec. 5.6, we perform a detailed comparison of our work against these prior works. As mentioned

in Sec. 4.1, DeepCache does not take advantage of frame-wise data reuse opportunities (e.g., reuse the inference result for similar frames). Thus, the latency improvement and energy saving potential of DeepCache can be limited. Additionally, as the step-size of full-inference is fixed, it cannot adaptively update its cache based on the video content. On the other hand, Euphrates makes use of the motion information collected from the Image Signal Processor (ISP), and search the RoIs by combining the MVs of the current frame with the inference result of reference frame, and consequently, decrease the number of inference. Different from these two prior works where approximation decisions are dictated by reuse distance (e.g., the distance between two fully-inferenced frames), Potluck [36] utilized the feature vector [60] extracted from input frames to adaptively trade off computation with reuse of the cached results.

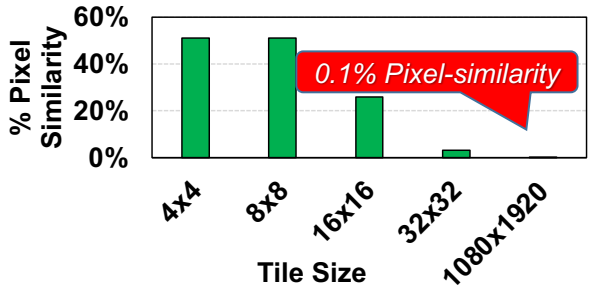
In addition to those described above, several offloading-based techniques have been proposed to speed up the inference on edge. For example, a dynamic RoI encoding is proposed to compress the data volume to be transferred through network [44]. An anchor-point selection algorithm (running in cloud) is also proposed to decide whether to reuse the previous results or not in [61]. Both require the assistance of the cloud, and hence, introduce additional costs and privacy issues.

To summarize, even though the prior works discussed above tried to optimize the DNN models, the static compiler/runtime support and hardware-based enhancements for performance- and energy-efficient execution of DNN pipelines for video data on mobile devices are still open problems. Our main goal in this work is to look deeper into the existing pipeline to dynamically identify better opportunities for optimizations, with minimum changes to the existing software-hardware stack.

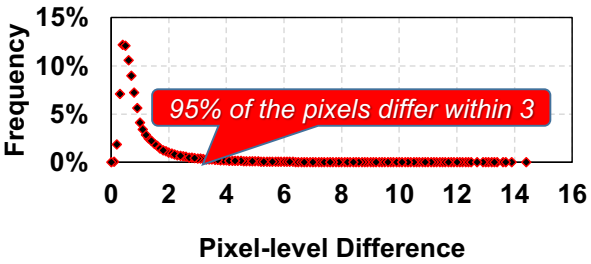
3.3 Similarity in Video Applications

Due to the limited computing resources available in mobile devices and strict power and energy budget constraints of mobile batteries [19], enabling high-quality fast inference is very challenging for DNN applications such as video analysis or object detection. In fact, the inference phase of VGG-16 [62], which is a popular DNN model, takes 240 *ms* to execute on an embedded Adreno 640 [63] edge GPU, which is far from *real-time*. Moreover, most of the existing solutions targeting such DNN applications demand specialized hardware and/or custom compiler support, and it is not straightforward for a developer to deploy them without multi-domain expertise across the optimization, compilers and hardware spectrum.

However, in the specific context of video applications, the "temporal continuity nature" of the video data presents itself as an opportunity that is yet to be fully exploited. As a sample work in this direction, Euphrates [30] employs motion vectors to detect insignificantly small changes between frames and skips unnecessary inferences. However, the number of skipped inferences there is *static*. We believe that, by looking deeper into the similarity between frames and identifying reuse opportunities at a "finer level", we can significantly reduce the number of inferences that have to be performed, thereby reducing the burden on the hardware. Further, if we can somehow make this opportunistic similarity exploitation more *dynamic* (i.e., the inference decision is made based on runtime contents), the solution can encompass most video streaming applications without affecting the current development and hardware stack.



(a) Similarity study at different tile size across adjacent frames for a video in VIRAT dataset [2].



(b) Distribution of pixel-level difference across two adjacent frames.

Figure 3.2: Similarity study.

As discussed in Sec. 4.2, most existing optimizations focus on accelerating the inference processing/computation only and not fully understand the underlying characteristics of the input data, some input-specific optimization opportunities could be easily missed in existing approaches. For example, in the mobile video inference application scenarios considered in this paper, the input data are continuous video frames, and can potentially contain rich "similarity" across different frames.

To explore these opportunities, Fig. 3.2a plots the "pixel-level similarity" between two successive frames in a video [2]. Here, we vary the tile size (an N×N tile is a block N×N pixels) from 4×4 to the entire frame (1080×1920) on the x-axis, and the y-axis gives the fraction of "identical tiles" (when compared pixel-by-pixel) in *two successive frames*. From these results, one can make the following observations: **1).** As we can see from Fig. 3.2a, small-size tiles share significant similarities across consecutive frames. E.g., nearly 50% of the tiles are identical in successive frames with a tile size of 8×8. **2).** However, as the tile size increases, tiles in successive frames become more dissimilar. E.g., when using 32×32 tiles, only 3% of the tiles in successive frames are identical. In the extreme case where the tile size is set as frame size, only 0.1% of the tiles in successive frames are identical.

As a result, any approach trying to exploit frame reuse (e.g., skip inferences for similar frames) based on this specific similarity metric (exact pixel match) will not have much scope for optimization. Instead, one may want to consider alternate similarity metrics that can lead to richer reuse opportunities.

The distribution of the differences (deltas) between the absolute pixel values in successive frames plotted in Fig. 3.2b provide one such opportunity. It can be observed that, while the "exact pixel match based similarity" is scarce at frame level, an alternate similarity based on the "magnitude of pixel differences" is abundant. Specifically, about 95% of the deltas (pixel differences in successive frames) are less than 3. The rest of this paper presents and experimentally evaluates two novel optimization strategies that exploit this similarity.

3.4 Proposed Strategies

As discussed in Sec. 6.3, there exist significant similarities between two successive frames (e.g., for more than 95% of the pixels between two adjacent frames) when considering "pixel difference" as the similarity metric. In the following two subsections, targeting inference-based video applications, we present two novel schemes that take advantage of this similarity: *frame-level pruning* and *region-level pruning*.

Algorithm 1: Adaptive Frame Level Reuse Algo.

Input : RD: Reuse Distance
Input : BBs: Bounding Boxes in Previous Frames
Input : MVs: Motion Vectors for Current Frame
Output : $Do\_Full\_Inference$: Decisions

procedure $Moving(BBs,\ mv)$ // Scenario1
  if $0 < max\{overlapped.area\} \leq T_{moving} \times mv.area$ **then**
    return $True$
  else
    return $False$

procedure $Missed(BBs,\ mv)$ // Scenario2
  if $mv.area \geq T_{missed} \times min\{BBs.area\}$ **then**
    return $True$
  else
    return $False$

procedure $Entering/Exiting(mv)$ // Scenario3
  if *mv is at edge* **then**
    return $True$
  else
    return $False$

procedure $Frame\_Decision(RD,\ BBs,\ MVs)$ // main
  if $RD \geq RD_{upper\_bound}$ **then**
    return $True$
  if $MVs == \varnothing$ **then**
    return $False$
  Fuse and Filter the MVs
  if $BBs == \varnothing$ ***and*** MVs_is_large **then**
    return $True$
  else
    return $False$
  for mv ***in*** MVs **do**
    if $mv.area > T_{area} \times minBBs.area$ **then**
      if $Moving(BBs,mv)$ ***or*** $Missed(BBs,mv)$ **then**
        return $True$
    if $Entering/Exiting(mv)$ **then**
      return $True$
  return $False$

3.4.1 Frame-Level Pruning

3.4.1.1 Shortcomings of Pixel-by-Pixel Comparison

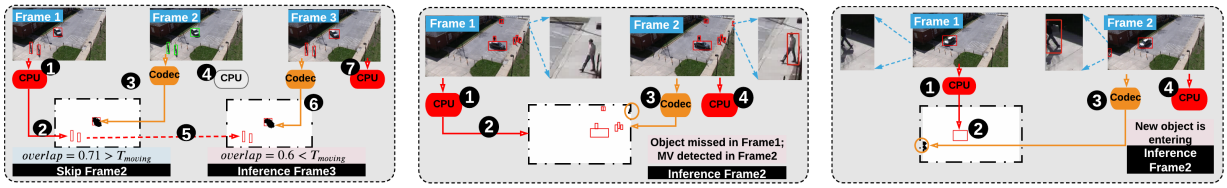
We believe a solution based on exact pixel-by-pixel matching would be too strict and would not be suitable for neural network-based applications. This can be illustrated with a simple use-case scenario. Consider an object detection scenario where a multi-color LED bulb on a Christmas tree identified as an object of interest, glowing red in Frame_i, changes to blue in Frame_{i+1}. In such a case, as the pixel values of the identified object have changed, Frame_{i+1} cannot reuse the identification result (LED bulb) from Frame_i even though it should ideally be able to do so in an object detection application. Hence, rather than relying on low-level raw pixel values, most neural network applications leverage high-level features, where "new events" or "motions" make more sense to employ in determining whether we can reuse the results from previous frame (or skip the current frame). We next discuss the motion vector concept, which is used in our approach.

3.4.1.2 Motion Vectors

The motion vector (MV) concept, which is built upon pixel differences, can be a good candidate to capture the reusability in video analytics. A MV is a two-dimensional vector that provides an offset from the coordinates in the decoded frame to the coordinates in a reference frame [64], which can be directly obtained from the codec hardware [65] without the need for any post-processing. As opposed to the software-based "optical flow" solution widely used in the computer vision domain [66], collecting the MV from the *codec hardware* is quite light-weight. In fact, our profiling indicates that only tens of μs are needed to generate the MVs for one frame (negligible compared to milliseconds or even seconds required for DNN inference). We next illustrate 3 common scenarios of leveraging the MVs to capture the reuse opportunities in a DNN-based object detection application.

3.4.1.3 Three Scenarios of Frame-Level Reuse

ⓘ **Moving Object(s)**: As shown in Fig. 3.3a, one of the most common cases in videos is that one or more objects (which have been identified in previous frames, i.e., Frame-1) move around in the current frame (i.e., Frame-2 , Frame-3). In such scenarios, to explore the reusability exposed by motion vectors, ❶ we first process the full inference in CPU for Frame-1, and ❷ identify the objects as well as their positions (i.e., bounding boxes [67]).



(a) Scenario 1: moving. (b) Scenario 2: capturing a previously missed object. (c) Scenario 3: entering/existing.

Figure 3.3: Three scenarios: Scenario1: An existing object is moving. Scenario2: An object was not captured by the previous frame, but captured by the current frame. Scenario3: A new object is entering the frame (or an existing object is exiting).

❸ Then for Frame-2, we obtain the motion vector of this frame from the codec (with a minimal overhead, as discussed in Sec. 3.4.1.2). With such knowledge, for Frame-2, we can observe that the bounding box and the motion vectors mostly overlap (with an overlap ratio of 71% in the left case), indicating that the objects have barely moved. Thus, we can safely *skip* the whole inference process for Frame-2, and simply reuse the result from Frame-1, as shown in ❺. However, for Frame-3, the motion vector generated by codec (❻) drifts away from the bounding box in Frame-1 (with an overlap ratio of 60% in the right case), which indicates that the object has moved/shifted a significant distance. Thus, we need to perform *full inference* for Frame-3 to maintain high accuracy for detection, as shown in ❼. Putting all these together, in this scenario, both Frame-1 and Frame-3 employ full inference, whereas the inference for Frame-2 can be skipped, with very little overhead (only 0.5% of the time required for a full inference for the YOLOv3 model).

ⓘⓘ **Missing Object(s)**: Another scenario is one in which one or more objects, which have not been identified in the previous frames, are detected in the current frame, as shown in Fig. 3.3b. In this scenario, the area of the MV S_{mv} is similar to or even larger than the area of the smallest bounding box S_{sbb} from the previous frame (in this case, $S_{mv} = 1.3 \times S_{sbb}$). Such large MV indicates that an object (of similar size to that of the other objects which have been already identified), which was supposed to be captured, but failed due to the inaccurate light-weight DNN models (such as YOLOv4-tiny [47] used by the detection application as discussed in Sec. 5.6). Therefore, in this case, Frame-2 needs to be carefully processed and we need to employ full inference for it, as indicated by ❹ in Fig. 3.3b.

ⓘⓘⓘ **Entering/Exiting Object(s)**: Another scenario is where one or more objects are moving into or out from a frame (refer Frame-2 in Fig. 3.3c). In this case, although the

MV is smaller than the bounding boxes, its position is on the edge, indicating that a new object is entering the frame. As a result, Frame-2 is critical as it reveals new information that has not been exposed before, and thus requires performing full inference for it.

3.4.1.4 Algorithm

To handle the three common scenarios discussed above, we propose our *adaptive* frame level (FL) reuse algorithm to determine whether to invoke full inference (FI) or skip inference (SI) for the current frame – as shown in Line `16` in Algo. 1. First, to ensure minimal accuracy loss, if we have already skipped inference for frames beyond a certain number ($RD_{\text{upper\_bound}}$), we decide to opt for FI (in Line `17`). Note that $RD_{\text{upper\_bound}}$ can vary across different use cases from parking lot to busy roads and hence, can be set accordingly for different cases. After filtering out the "no change in MV" cases for SI in Line `19` in Algo.1, we next fuse all of the MVs together and filter the small ones (i.e., noises) out (shown in Line `21`). Next, we iterate each MV block for Scenario-1, Scenario-2 and Scenario-3 (shown from Line `22` to Line `31`). More specifically, if the MV block is relatively large, then we examine whether the object is moving in the current frame or has been missed by the previous frame (in Line `27-28`). If that is the case, then full inference needs to be invoked. Otherwise, we further check whether an object is entering to or exiting from the current frame (in Line `30`) to perform full inference for it. Our experiments reveal that the proposed FL scheme can skip up to 53% of the frames, with *very less* accuracy loss (0.075%). Moreover, the total overhead of this algorithm is only 0.5% of the overall inference execution latency even for a heavy model, as will be discussed later in detail in Sec. 5.6.

3.4.2 Region-Level Pruning

As discussed in Sec. 3.4.1, while our proposed frame level scheme (coarse granularity) enjoys FI skipping opportunities in some cases, we need to perform FI for the remaining cases, to maintain the accuracy. This leads us to our next question – **Can we do better?** To explore the reuse opportunities at a finer granularity, we now dive into a tile/region-level study.

3.4.2.1 Partial Inference

To further explore the computation reuse opportunities, first, we revisit the first scenario illustrated in Fig. 3.3a, where the MV for Frame-3 is not highly overlapped with the

bounding boxes from Frame-1 (overlap ratio = 0.60, lower than the predefined threshold). This means that the object in question has been moving towards a direction, which triggers a "position change" event. Consequently, to reflect this event, Frame-3 needs to perform a full inference. However, as we can see from ❻ in Fig. 3.3a, only the bounding boxes (in red) and the MV (in black) are meaningful for the inference, and the remaining regions (in white) contain the same information as the previous frames, which, intuitively, do not provide as much contribution to detect the "position change" event denoted by the colored regions. This observation reveals an opportunity to perform *partial inference (PI)* by operating only on the bounding boxes and MV regions. Next, we study the following three questions in detail: (i) for which frames can we opt for the PI?, (ii) how to maintain the accuracy (with minimum loss)?, and finally, (iii) how to do the PI?

3.4.2.2 High-level Idea

Algorithm 2: Region Level Reuse Algo.

Input : RD: Reuse Distance
Input : BBs: Bounding Boxes in Previous Frames
Input : MVs: Motion Vectors for Current Frame
Output : $Flag$: Decisions (Full, Partial, or Skip)
Output : $RoIs$: Regions of Interest

```
1 procedure Region_Decision(RD, BBs, MVs)                    // main
2     if Frame_Decision(RD, BBs, MVs) is False then
3         return {Skip, null}
4     if IsScenario2 or IsScenario3 then
5         return {Full, null}
6     for mv in MVs do                          // only consider Scenario1
7         if max{overlapped.area} ≤ T_moving2 × mv.area then
8             return {Full, null}
9     return {Partial, ⋃[BBs, MVs] }
```

To answer the above questions, we propose region level partial inference (PI) to first determine whether one frame should process the PI, and if so, reuse the memorized computation results for "unimportant" regions and only process the inference for the "regions of interest" (bounding boxes and MVs), to save computations without much accuracy loss.

To explain the high-level idea behind PI, we reconsider Frame-3 in Scenario-1 discussed

in Sec. 3.4.1, and present our five-step solution in Fig. 3.4. In Step ❶, same as our FL scheme discussed in Sec. 3.4.1, the bounding boxes (BBoxes, in red) are extracted by the "full-inferenced" previous frame, and the MVs are obtained from the current frame (Frame-3). Next, based on these inputs including the BBoxes as well as MVs, Algo. 2 is invoked to decide how to process this frame (FI, PI, or SI). As one can see from Line 2 of this algorithm, if the frame has been labeled as "Skip" by Algo. 1, it can be safely bypassed. Additionally, if the frame is in either Scenario-2 or Scenario-3, it indicates that the object(s) in the current frame are different from the last inference outputs (i.e., "missed" in Scenario-2, or "entering/exiting" in Scenario-3) and hence, requires full inference (refer to Line 5 in Algo. 2). Otherwise, as can be seen from Line 6 to Line 8 in Algo. 2, the overlapped area of each MV and BBox is examined to determine whether the object has moved "too far away" or not; if it has, the "FI" is triggered. Finally, if the frame falls under the category in which objects have not been significantly displaced, then it is labeled as "PI" (as shown in Line 9 in Algo. 2), and the union of the BBoxes and the MVs is reported as the *new* regions of interest (RoIs) (see ❷ in Fig. 3.4). With these new RoIs, in Step ❸, we now only need to focus on the inputs that map to the RoIs, and omit the other "unimportant" regions. Next, the new inputs are fed into Step ❹ to perform the PI (details are in Sec. 3.4.2.4), and finally, we report the output result to the application, as shown in the blue BBoxes in Step ❺.

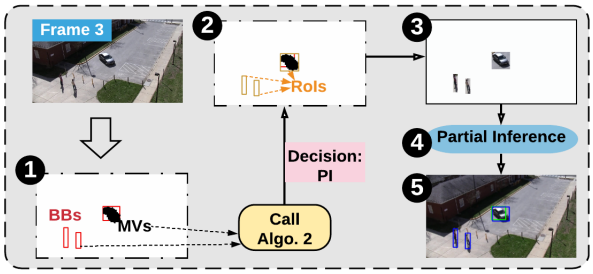


Figure 3.4: Main idea in our proposed partial inference scheme.

Since Algo. 2 has answered the first question (i.e., for a specific frame, what is the inference choice – FI, PI or SI?), we now turn to the second question raised above – how to maintain accuracy? – and study the layers in DNN to explore how to identify which parts are important and which are not.

3.4.2.3 How to Maintain Accuracy?

To preserve accuracy, we back-trace the output feature map (FM) to investigate how the different regions of the input FM affect the the output for convolution layer[1], as shown in Fig. 3.5 and carefully consider our design decisions. Here, we partition the output FM

[1]Since CONV layers dominate total computation in DNN inference [1], our partial inference technique is applied only to the CONV layers; the other layers employ full inference.

regions into three categories – ⓐ inner part, ⓑ middle part, and ⓒ outer part. ⓐ is the region where the convolution kernel only multiplies with the pixels in RoIs. In ⓑ, the kernel is multiplied with both the RoI pixels and the non-RoI pixels (background region, i.e., BG), and finally, ⓒ is where the kernel is only multiplied with the pixels inside BG region. Thus, the inner part of the output is only related to the RoIs of input; the middle part is related to both the RoIs and the BG; and the outer part is only related to the BG.

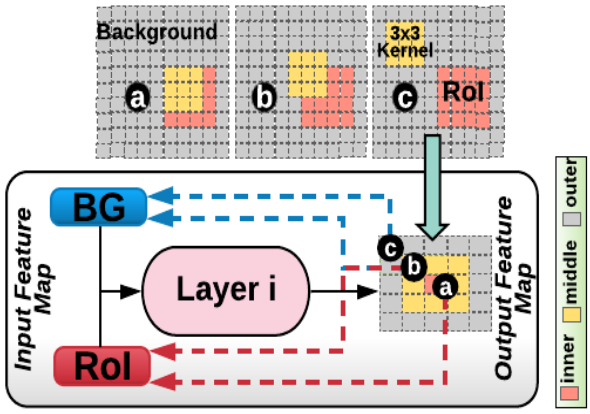


Figure 3.5: Back-tracing from *out* to *in* for CONV.

As discussed in Sec. 3.4.2.2, the RoIs are essential to the output results. Thus, the inner part has to perform full inference. However, it is *not* desirable to execute the inference on the middle part, since, in order to do so, one needs to prepare an input larger than the inner region (due to the various kernel sizes in the convolution, e.g., 3×3, 5×5 [68]), which eventually turns out to be the FI case. Rather than computing, we *memoize* the FM of the middle part in the previous frame, and reuse the data for the current frame. Since the outer part maintains no information for the RoIs, it can be safely discarded.

3.4.2.4 How to Do Partial-Inference?

Next, we revisit the example scenario discussed in Fig. 3.3a, and give the details of the proposed partial inference (PI) scheme in Fig. 3.6. Recall that, Frame-1, as the base frame, has to do the full inference (Step ❶), and output the feature maps for each layer (Step ❷). As for Frame-3, only the RoIs (in purple) are passed to the convolution layers (Step ❸), and we generate the corresponding partial-inference-outputs (in yellow) in Step ❹.

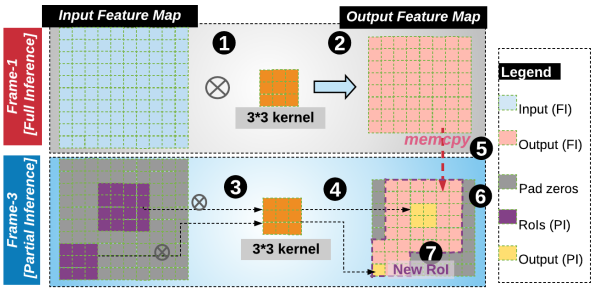


Figure 3.6: Details of the partial-inference for Frame-3 in Scenario-1.

Next, as discussed in Sec. 3.4.2.3, the full inference output (the neighbors of the partial inference outputs – pink color) from the

previous frame is critical to maintain the accuracy of the output for the current frame. Thus, the full inference outputs are accordingly padded around the partial inference outputs via memory copy (Step ❺). For the other regions in the output feature map for this frame, it is safe to simply pad them with zeros (Step ❻). And, finally, we update RoI for the next layer (the region with the purple border in Step ❼). The execution time of the PI for Frame-3 is only 54% of the FI, due to a large amount of computation reduction (achieved by omitting the unimportant regions).

3.4.3 Design and Implementation

We designed our both schemes (frame-level reuse in Sec. 3.4.1 and region-level reuse in Sec. 3.4.2) as plugable modules to the existing compute engines (e.g., CPU, GPU, etc.), as shown in Fig. 3.7. A *Decision Maker* is placed before the original compute engine (CPU in this example) to dynamically decide how to process the incoming frame, and opportunistically bypass the inference computation, i.e., either ❶ Full Inference (FI) or ❷ Skip Inference (SI), or reduce the compute by only processing a small part of regions, when opted for ❸ Partial Inference (PI), with a little overhead (0.5% w.r.t. DNN inference for YOLOv3). To implement these three possible decisions (FI, SI, and PI) that can be made by the *Decision Maker*, the *Inference Engine* in Fig. 3.7 takes the corresponding actions:

Full Inference: When the incoming frame contains critical information such as new objects entering, a full inference is needed. In this case, the current frame is processed on the CPU to report the final result. Apart from that, the generated bounding boxes for the object detection task and the feature maps for each layer during the inference are intermediately stored in memory as a "checkpoint". By referring this, it can potentially benefit an inference to be performed later by either completely skipping it (SI) or reducing the computation (PI).

Skip Inference: When the current frame is identified as "clonable" by the previous frame, the CPU is released without any inference execution request. Instead, the previous results (the bounding boxes [including the classes and scores] in the previous frame), which have been memoized before, are directly loaded as the inference result for the current frame.

Partial Inference: In this case, rather than processing the whole frame, only the RoI blocks are fed into the CPU and, with the memoized feature maps from previous frame, the CPU is able to generate the desired result for the current frame as accurate as performing inference on a full frame.

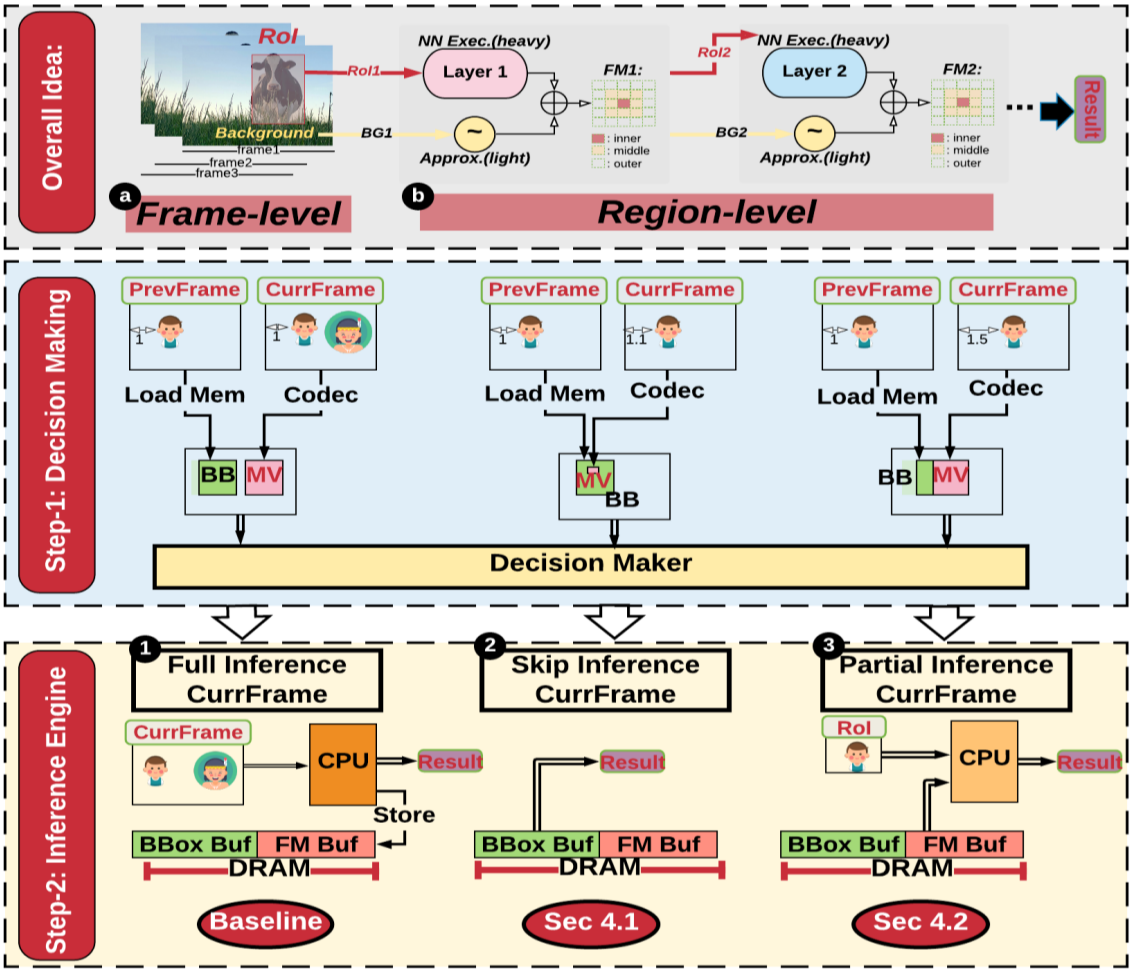


Figure 3.7: The proposed frame-level reuse and tile/region-level reuse design blocks implementation; BB/BBox: bounding boxes from the last FI; MV: motion vectors of the current frame; FM: feature maps for each layer from the last FI.

3.5 Evaluation

Targeting the object detection task on edge devices, we compare our proposed frame-level scheme (*Full-Inference* and *Skip-Inference*, i.e., FI+SI) and region-level scheme (*Full-Inference*, *Skip-Inference* and *Partial-Inference*, i.e., FI+SI+PI) against the *baseline inference*, which performs full inference on every frame, as well as four state-of-the-art runtime techniques (DeepCache [1], Euphrates [30], Potluck [36], and MCDNN [37]), by quantifying the normalized execution time, energy consumption, and accuracy (mean Average Precision, mAP). We first describe the design configurations, experimental platform, datasets, and measurement tools used in this work, and then analyze the collected results.

3.5.1 Design configurations

Baseline: We evaluate the baseline video object detection on an edge CPU, where every frame is fully inferenced.

FI+SI: We evaluate our FI+SI scheme that skips the inference computations by utilizing MVs captured by codec, as discussed in Sec. 3.4.1. The first frame is considered as the "base", which is always fully inferenced. For the remaining frames however, we invoke Algo. 1 to make the frame-level decision (i.e., either do the full inference or skip)[2]. When skipping, we can simply bypass the inference task for this frame by *reusing* the output results from the last frame; otherwise (i.e., if cannot skip), the full inference has to be performed for the current frame, in the same fashion as in the baseline.

FI+SI+PI: Different from the above FI+SI scheme, in this scheme, the region level decision-making Algo. 2 is invoked to decide among the execution choices for the incoming frames – including Skipping, Full-Inference, and Partial-Inference.

3.5.2 Experimental Platform and Datasets

Platforms: We used the Google Pixel 3 Android Phone [45] as our experimental platform. This device consists of a 64-Bit Octa-Core, a Qualcomm Snapdragon 845 SoC, [69], a 4GB LPDDR4X memory, and a 64GB storage. We implemented our FI and PI on top of the ncnn library [27].

mean Average Precision (mAP): In object detection applications, the accuracy of the DNN models is usually quantified by mAP [70], which is a metric to evaluate how well the estimated detection results match the ground-truth. To get mAP, the precision for each class is first calculated across all of the Intersection over Union (IoU [71]) thresholds, then the final result is generated with the averaged precision for all classes (the higher, the better). The IoU threshold is set as 0.5 in our evaluations, which is a widely used value in mAP calculation.

Datasets: We use three published video datasets (VIRAT [2], EPFL [72] and CAMPUS [73]), to study the performance and energy behavior across different videos. The important features of these six videos[3] are summarized in Table 3.2.

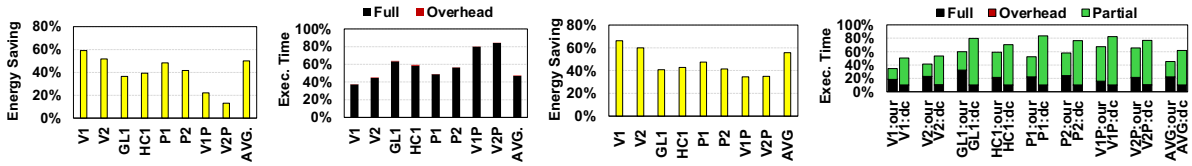
Neural Network Models: We examine two DNN models in our experiments: YOLOv3 [46] and YOLOv4-tiny [47]. The input shape for both of them is $1 \times 416 \times 416 \times 3$. The

[2]We experimentally set $T_{moving} = 0.8$, $T_{missed} = 0.8$, $T_{area} = 0.25$ and $RD_{upper\_bound} = 10$ in Algo. 1; $T_{moving2} = 0.3$ in Algo. 2

[3]The ground-truth annotations for the EPFL and CAMPUS datasets were not available to us; so, in this work, we primarily focused on the VIRAT dataset to evaluate the accuracy.

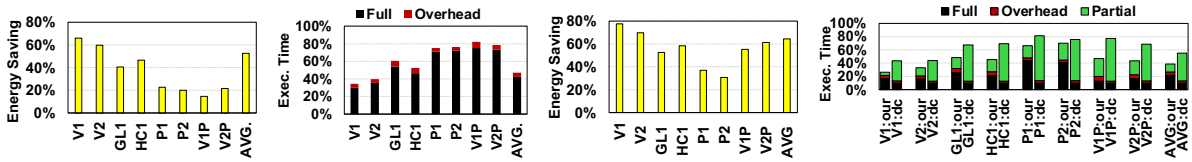
Table 3.2: Salient features of the six videos used in this study.

Videos	# Frames	# Avg. Objects	Static/Dynamic	Object/Full Frame	Description
V1 [2]	20655	Medium	Medium	Small	A few people and cars move in the parking lot
V2 [2]	9075	Medium	Medium	Small	More activities of people in the parking lot
GL1 [73]	6477	More	Dynamic	Medium	More activities of people and cars in the parking lot
HC1 [73]	6000	More	Dynamic	Medium	People do exercises in a garden
P1 [72]	3915	Less	Dynamic	Small-Large	Several people walk around in a room
P2 [72]	2955	Medium	Dynamic	Small-Large	More people than P1 in the room



(a) FI+SI energy saving (b) FI+SI exec. time (c) FI+SI+PI energy saving (d) FI+SI+PI exec. time

Figure 3.8: Performance and energy improvements for YOLOv3 w.r.t. the baseline; `XX:our` are the results achieved by our proposed FI+PI+SI scheme; `XX:dc` are the results for DeepCache [1]; `V1P` and `V2P` are part of V1 and V2, respectively; and `AVG` is the weighted average (weight is the # of frames in each video).



(a) FI+SI energy saving (b) FI+SI exec. time (c) FI+SI+PI energy saving (d) FI+SI+PI exec. time

Figure 3.9: Performance and energy improvements for YOLOv4-tiny w.r.t. the baseline; Region Level Reuse Scheme (e.g., FI+SI+PI) has better performance, because this "shallow" model benefits more from partial inference.

former one is a quite heavy model, with 106 layers and 65.86 Bn FLOPS, whereas the latter one is a lighter model, with 38 layers and 6.94 Bn FLOPS. Note that, *YOLOv3 achieves state-of-the-art accuracy in object detection*, with an average mAP of 55.3%, whereas YOLOv4-tiny achieves 40.2% [74] mAP.

3.5.3 Results

We present and compare the execution time and energy consumption (via BatteryManager API in Android Studio) when performing inference for each video under the three configurations described in Sec. 3.5.1, as well as the mAP to evaluate the accuracy, and

Table 3.3: mAP comparison with the baseline

	V1	V2	V3	V4
YOLOv3	51.8	55.4	42.0	59.5
YOLOv3 with FI+SI	51.8	55.4	41.8	59.4
YOLOv3 with FI+SI+PI	50.3	54.1	40.6	58.0
YOLOv4-tiny	31.8	49.3	31.3	33.5
YOLOv4-tiny with FI+SI	31.6	49.2	31.4	33.3
YOLOv4-tiny with FI+SI+PI	31.4	49.1	30.8	33.0

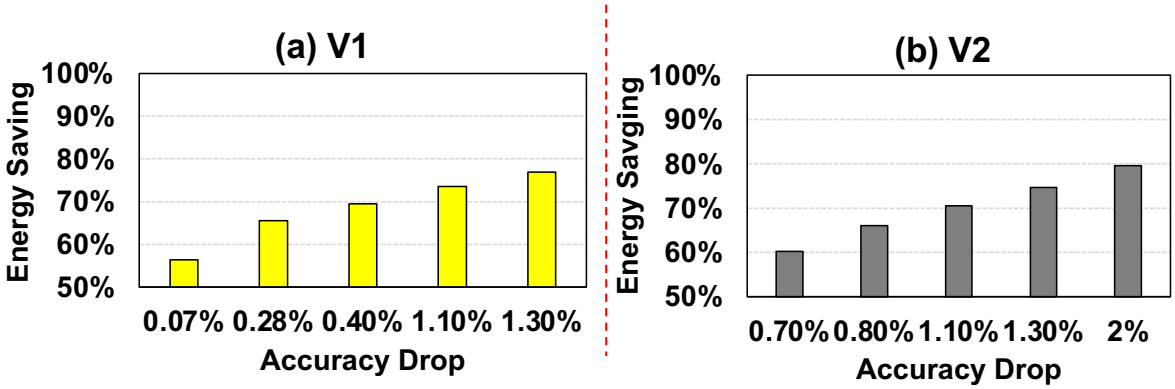


Figure 3.10: The tradeoff between accuracy drop and energy saving for (a) V1 and (b) V2 picked from the VIRAT [2] dataset. This shows how the proposed "adaptive" solution can potentially save more energy with different thresholds in Algo. 2.

plot the experimental results in Fig. 3.8 and Fig. 3.9. Note that these execution time and energy results are *normalized* with respect to the baseline (full inference for all frames). From these results, we can make the following observations.

3.5.3.1 Overall Execution Latency

On average, our FI+SI scheme can reduce 52% of the execution time for YOLOv3, and 53% for YOLOv4-tiny, compared to the baseline. On the other hand, our FI+SI+PI scheme can save 55%/61% of the execution time for YOLOv3/YOLOv4-tiny. More specifically, as shown in Fig. 3.8b and Fig. 3.8d, the overheads (due to the decision making module in Algo. 1 and Algo. 2) introduced by our proposal amount to only 0.5% of the end-to-end execution latency (the baseline YOLOv3 inference). Also, the PI technique consumes 23% of the execution time in YOLOv3.

Similarly, the overhead introduced by Algo. 1 to YOLOv4-tiny is not too much – 5% of the baseline execution latency, as shown in Fig. 3.9b. Such small amount of overhead indicates that our decision-making logic is efficient and light-weight.

3.5.3.2 Energy Savings

As shown in Fig. 3.8a, the YOLOv3 inference with the proposed FI+SI scheme only consumes 50% energy on average, with respect to the baseline, due to the fact that 53% of the inferences can be skipped. The PI technique can further save 6% more energy on average, as shown in Fig. 3.8c. On the other hand, the YOLOv4-tiny inference with the FI+SI scheme saves 53% energy w.r.t. the baseline, as shown in Fig. 3.9a. Moreover, when adopting the FI+SI+PI, compared with the FI+SI, around 12% more energy can be saved, as shown in Fig. 3.9c. These additional energy savings come from the computational reduction by partial inference, rather than computing for the entire frame.

3.5.3.3 mAP

To study the accuracy impact of our proposed SI and PI schemes, we summarize the mAP for our two models (YOLOv3 and YOLOv4-tiny) using four videos from VIRAT dataset [2] in Table 3.3. One can observe that the mAP in our FI+SI scheme only drops by 0.075% on average for YOLOv3, and 0.1% for YOLOv4-tiny – both w.r.t. the baseline. With the FI+SI+PI scheme on the other hand, the mAP drops 1.4% in YOLOv3, and 0.4% for YOLOv4-tiny. These results clearly show that our proposal is able to maintain high accuracy.

3.5.3.4 Model-Specific Analysis

To study how the inference behavior changes across different DNN models, we next compare the performance and the energy consumption of two DNN models used in this work (YOLOv3 and YOLOv4-tiny), and plot the results in Fig. 3.8 and Fig. 3.9. First, by comparing YOLOv3 (see Fig. 3.8b) and YOLOv4-tiny (see Fig. 3.9b), we can observe that YOLOv4-tiny spends less time on inference than YOLOv3 (42% versus 47%). However, the overhead this scheme brings when applied to YOLOv3 (0.5% w.r.t. the baseline) is smaller than that in YOLOv4-tiny (4.8%), resulting in a similar reduction in the overall execution time. Note that, the reason why the overhead in YOLOv4-tiny is higher is because, YOLOv4-tiny is already a very light-weight model, and consequently, the time spent on decision making is not negligible compared to the extremely fast inference it performs.

Additional energy and latency can be saved by the FI+SI+PI scheme, as shown in Fig. 3.8c-3.8d and Fig. 3.9c-3.9d, i.e., 55% latency and 56% energy saving in YOLOv3, and 61% latency and 64% energy saving in YOLOv4-tiny. The reason why YOLOv4-tiny

saves more is that the PI benefits more in a "shallow" model with a relatively larger room to skip. For example, YOLOv3 takes, on average, 0.75× of the baseline latency to perform the PI on a frame in video HC1 [73], whereas YOLOv4-tiny takes only $\sim 0.48\times$ of the baseline latency.

3.5.3.5 Video-Specific Analysis

To investigate how the performance and energy behavior varies with video salient features such as the object count, objects' motions, the fraction of area occupied by objects in an entire frame, etc., we have studied the six videos summarized in Table 3.2. V1 and V2 are parking lot videos. V1 has fewer objects and less movements, and thus, compared to V2, the savings on execution time and energy consumption for V1 are slightly higher, as shown in Fig. 3.8b and Fig. 3.8a. Also, V1 can save 7% more execution time and 6% more energy than V2 when using the FI+SI+PI scheme.

GL1 and HC1 are other two videos from the CAMPUS [73] dataset. GL1 is also a parking lot video, but with many more objects and more frequent movements than V1 and V2. HC1 on the other hand is a garden video, with people walking around, riding bike, etc. These two videos are much more active (dynamic) than V1/V2, which means the reuse opportunities they contain are not as much as V1/V2. Thus, compared to V1 and V2 whose latencies can be decreased by 62% and 55% respectively with FI+SI as shown in Fig. 3.8b, the latencies for GL1 and HC1 only decrease by around 38%, while the energy saving is about 37%, which is lower than the corresponding savings for V1 (59%) and V2 (53%). Additionally, even with our proposed PI technique, the improvements for GL1 and HC1 with YOLOv3 model are not very significant (e.g., the latency reduction is improved by 4% for GL1 as shown in Fig. 3.8d). This is because, compared with the whole frame, RoIs in these two videos are large (e.g., ratio of RoI/whole frame is 4x of that ratio in V1), which increases the computation for the PI and thus decreases the its benefits.

Finally, P1 and P2 are two videos with several people walking around a room. Since the RoIs size increases along the video (e.g., ratio of RoIs/whole frame increases from 8% in frame 20 to 53% in frame 2470), the benefits from PI become increasingly lower, resulting in similar patterns for FI+SI and FI+SI+PI, in YOLOv3 model. Besides these six videos, we also picked two *slices* from V1 and V2 (called V1P and V2P) to demonstrate the effectiveness of our proposed PI technique. In these two slices, the objects keep moving. Thus, most of the frames will be performing full inference under FI+SI, as shown in Fig. 3.8b. The latency is only decreased about 18%, while the energy

saving is 22% and 13%, respectively, as shown in Fig. 3.8a. When employing the PI scheme, as shown in Fig. 3.8d and Fig. 3.8c, the latency is further decreased by 13% (V1P) and 19% (V2P), whereas the energy saving is increased by 12% and 22%. The improvement brought by PI for YOLOv4-tiny model is more significant as shown in Fig. 3.9c and Fig.3.9d, where the latency is further decreased by 36% (V1P) and 35% (V2P), while the energy saving is increased by 41% and 40%, indicating that our PI technique is quite effective.

3.5.3.6 Tradeoffs between Accuracy and Energy Consumption

So far in our evaluation, we wanted to minimize the accuracy impact (see Table 3.3). However, as an alternate design principle, one may want to relax this accuracy constraint and thus save more energy. We used Pytorch [75] to profile the accuracy behavior of two videos picked from VIRAT [2] dataset, and show that our proposal is able to adaptively support such alternate design choices. More specifically, the <accuracy-loss, energy-savings> pairs are plotted in Fig. 3.10. From Fig. 3.10(a), one can observe that, for V1, the energy saving is about 56% with only 0.07% accuracy drop; however, if the application is willing/tolerant to live with 2% accuracy drop, then 21% more energy can be saved (amounting to a total of 77% energy reduction compared to the baseline strategy). A similar trend can also be observed in V2, as illustrated in Fig. 3.10(b). These results clearly indicate that our proposal is flexible/adaptive with different design preferences.

3.5.4 Comparison against Prior Work

As discussed in Sec. 4.2, DeepCache [1] also exploits the similarity between continuous frames at runtime, and decreases the inference computation via memory copies. However, it does not take advantage of the opportunities for frame-level data reuse, and hence needs to perform inference for each and every frame. Thus, as shown in Fig. 3.8d and Fig. 3.9d, DeepCache can only save 38% and 45% on execution time for YOLOv3 and YOLOv4-tiny, respectively, which are less than both FI+SI (e.g., 52% for YOLOv3; 53% for YOLOv4-tiny) and FI+SI+PI (e.g., 55% for YOLOv3; 61% for YOLOv4-tiny) schemes.

Another hardware-based optimization has been proposed in Euphrates [30], as discussed in Sec. 3.2.2.3. We acknowledge that, compared to our proposal, Euphrates yields around 10% additional energy savings when the window size is set to be as large as 8 (i.e.,

always skipping 7 frames). We want to emphasize however that, such static (pre-defined) window-size works well only when there are few movements in the videos. On the other hand, when the video has more dynamic behavior, this static skipping jeopardizes the quality of applications which demand high accuracy. To give an example, we selected a segment of frames (i.e., `Frame#6750` to `Frame#6850`) from V1 [2], in which the objects move more aggressively than other segments. In this scenario, our scheme figures out that more movements exist from large MV blocks, and dynamically adjusts the inference decisions. Hence, our scheme does not lose any accuracy, and still saves 43% energy. However, due to the static window size employed, Euphrates [30] leads to as much as 6% accuracy drop in this video segment, which is hardly acceptable by most applications.

Apart from the above mentioned optimizations at the layer-level (DeepCache) and the frame-level (Euphrates), recall that, in Table 3.1, we indicated (in the last column) the Decision Making Logic (DM) the previously proposed optimization strategies employ. Now, in Table 3.4, we present the mAP, latency and energy efficiency results with those prior schemes. Rather than capturing the reuse opportunities in raw input data, Potluck [36] first extracts the feature vector (i.e., a vector generated from input image, such as SURF [76], HoG [77], Down-sampling [43], etc.) as *key*, and then caches the corresponding results (as *value*) for further reuse. We also conducted experiments with a strategy mimicking Potluck [36] on V1 [2] by employing the down-sampled image as the feature vector. As can be observed from Table 3.4, Potluck [36] provides very good accuracy – 51.6, which is slightly better than 50.3 provided by our approach. However, it results in almost twice the latency and energy consumption with respect to our approach. This high latency/energy consumption is due to the imprecise down-sampled feature vector, and this further indicates that, a strategy that is based on feature vector memoization can still miss a lot of optimization opportunities.

Different from the three prior works discussed above, which mainly focus on one model, MCDNN targets a multi-model system and proposes a runtime scheduler to improve the accuracy as much as possible within a limited energy budget. We also tested this idea in our framework (with YOLOv3 [46] and YOLOv4-tiny [47] as the available models) and set the energy budget for the scheduler to be the total energy consumption resulting from our approach. As can be seen from Table 3.4, MCDNN yields similar latency/energy savings compared to our approach, but it results in significantly lower accuracy (36.9). This is mainly because, in MCDNN, the scheduler tends to choose YOLOv4-tiny due to the low energy budget.

Table 3.4: Comparison against Potluck and MCDNN

	mAP	Latency	Energy
MCDNN	36.9	33%	35%
Potluck	51.6	63%	61%
This Work	50.3	35%	34%

3.5.5 Future Work

As shown in Sections 3.4.1 and 3.4.2, our inference decision (FI, SI, or PI) is made by comparing the overlap between the MVs and the previous frame's BBoxes, or the ratio of the MV blocks size to the previous BBoxes, with a preset thresholds. Although we have an intuitive feeling on the how the thresholds in Algo. 1 and Algo. 2 affect the accuracy and performance, e.g., a larger T_{moving} in Algo. 1 has a preference on FI and is accuracy-friendly, while decreasing T_{moving} will skip more frames and improve the performance, the specific impact of the selected thresholds on accuracy drop, performance improvement, and energy reduction deserves further study.

Also, as discussed in Sec. 3.5.3, the overheads brought by our decision making amount to 5% of the DNN inference time for YOLOv4-tiny when running on CPU, which reduces the performance and energy savings for our proposed schemes. If we can deploy the decision making logic on a *custom hardware* with negligible overhead, our proposed techniques would be more effective when targeting light DNN models.

3.6 Concluding Remarks

Pushing DNN-assisted video analysis into edge is the current trend in applications such as surveillance, VR/AR, assisted surgery, and synopsis extraction [50]. Along this trend, prior approaches have targeted improving either accuracy or performance. In contrast, this paper revisits the <accuracy, energy, performance> design space, and tunes the design knobs adaptively with the changing constraints of applications over time. Specifically, we propose and evaluate two schemes (for skipping the inference computation at frame-level, and bypassing the compute for unimportant regions of the video frame), to improve performance and save energy. Our collected experimental results indicate 2.2× performance speedup and 56% energy saving over the baseline setting. Furthermore, relaxing the accuracy loss tolerance to 2%, our energy savings can reach up to 80%. Additionally, the experimental analysis indicates that our approach outperforms the state-of-the-art work with respect to accuracy and/or performance/energy savings.

Chapter 4
Efficient HDR Content Generation on Edge Devices

4.1 Introduction

[78]

4.2 Background and Motivation

In this section, we xxxx

4.2.1 Background

4.2.1.1 High Dynamic Range Imaging

Dynamic range, which is defined as the ratio between the maximum and minimum luminance in the photography and imaging domain [79], is one of the key metrics that

(a) Generated HDR image using SingleHDR [] (b) Generated HDR image with our proposal (2X speedup) (c) Generated HDR image with our proposal (4X speedup)

Figure 4.1

characterize the quality of the visual contents. As depicted in [80], a higher dynamic range can retain more scene details. Typically, the range of luminance that human eye can handle is 10000:1 in a single view [79], for which the conventional RGB image with 8 bits per channel (also known as low dynamic range (LDR) image), struggles to capture. To bridge this gap and better represent real-world scenes, the high dynamic range (HDR) imaging technique was introduced and has been extensively explored during the past several decades []. Specifically, HDR imaging is a technique used to produce images or videos with high dynamic range. Unlike LDR images, which store the pixels using 8-bit integers, HDR images utilize 32-bit floating-point data to represent pixel values for each channel, resulting in 96 bits per pixel. Consequently, HDR images can represent a significantly broader range of luminance.

4.2.1.2 HDR on Edge Devices

Recent advancements of edge devices like mobile phones or VR/AR headsets have enable them the avability to capture and display hdr contents.

4.2.1.3 HDR Content Generation

HDR contents can be captured using specialized cameras [], however, these cameras are usually expensive and impractical for edge devices due to their inherent resource constraints [81]. An alternative approach for generating HDR content is to reconstruct HDR images from visual content captured in low dynamic range (LDR). This approach can be further categorized into multiple-exposure HDR and single-exposure HDR. Specifically, in the multi-exposure HDR approach, multiple SDR images are captured with different exposure values. These SDR images are then aligned, merged, and fused to generate the HDR images. This technique is commonly used for HDR mode in mobile devices by companies like Apple and Google, for their iPhone and Pixel phones, respectively [82, 83]. While multiple-exposure HDR can effectively generate HDR images for static scenes, it may produce ghost artifacts in the presence of large motions. Moreover, aligning the SDR images, which directly affects the quality of the generated HDR images, requires extra effort. Additionally, there may be legacy SDR content containing unreproducible historical scenes for which multiple exposure versions are unavailable. Single-exposure HDR, also known as inverse tone mapping [], constructs an HDR image using only a single SDR image and can overcome the above issues associated with multiple-exposure HDR. However, due to the limited information from a single SDR image, the generated HDR might be of low quality and/or lack details and texture, making it an ill-posed

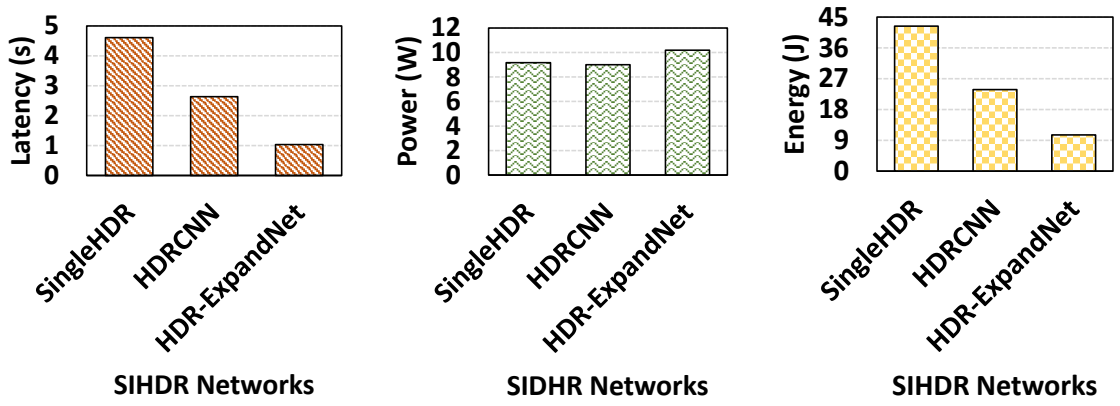
problem [84]. Fortunately, recent advancements in machine learning have introduced deep neural networks (DNNs) as effective inverse tone mappers for single-exposure HDR reconstruction [84–87]. However, these methods, while effective in generating high-quality HDR images, usually involve heavy DNN models and consume significant time and memory space for inference, as depicted later in Sec. 4.2.2.1. Given the unique resource constraints and limited battery budget of edge devices, using these DNN models to generate HDR images on edge devices remains a significant challenge. Although there are some prior attempts xx, however, they fail xx, as we will discuss in Sec. **??**.

4.2.2 Motivation

In this section, we first characterize the performance and energy behavior of the single-image HDR (SIHDR) DNNs on an edge GPU SoC, using three popular SIHDR DNN models in Sec. 4.2.2.1 and understand their inefficiencies. Following this, we explore the foveated nature of users' gaze direction when viewing the HDR visual contents in Sec. 4.2.2.2 and show the potential opportunities for optimizing HDR content generation by leveraging this foveated viewing in Sec. 6.3.2.

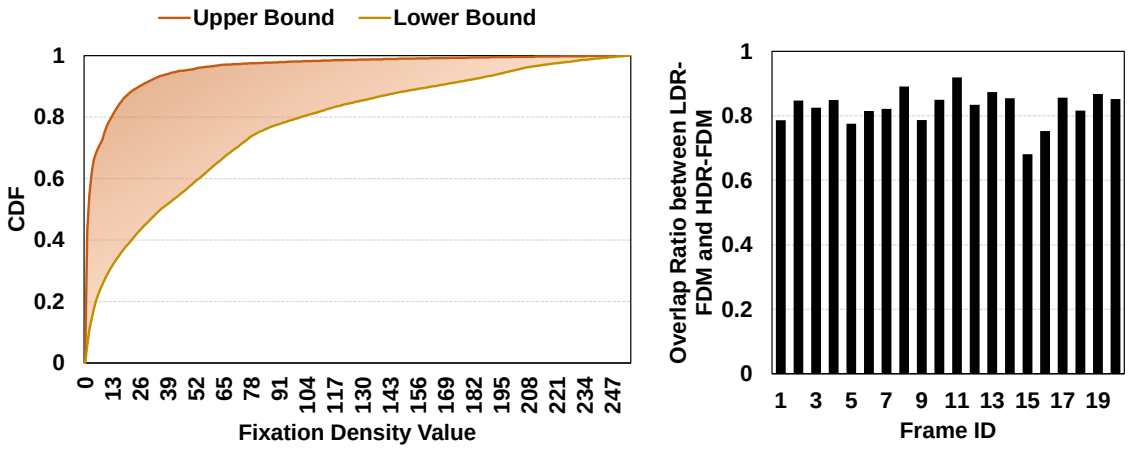
4.2.2.1 Performance Characterization for DNN-based SIHDR

To better understand the performance of the DNN-based HDR content reconstruction, we characterize the latency, power, and energy consumption of three widely-used Single image HDR DNN models, including SingleHDR [85], HDRCNN [87] and HDR-ExpandNet [86], on a typical edge SoC platform (NVIDIA Jetson Orin Nano [88]). As shown in Fig. 4.2a, overall, all three DNN models require at least a few seconds to perform the inference. This extended latency will lead to noticeable processing delays and significantly sacrifice the user's experience. Moreover, as illustrated in Fig. 4.2b, the power consumption for SingleHDR [85] and HDRCNN [87] is around 9W, with HDR-ExpandNet [86] consuming up to 10W. Considering that the typical use cases such as audio/video playback or phone calls generally consume no more than 1000mW power [89], this level of power consumption is substantially high for edge devices and could lead to faster battery drain and potential overheating issues. The combination of long inference latency and high power consumption results in exceedingly high energy consumption, as depicted in Fig. 4.2c, which is unacceptable for edge devices. For instance, consider a mobile phone with an average battery capacity of 4000mAh [**?**] and a voltage of 3.7volts [**?**], merely processing approximately xx frames of inference using HDR-ExpandNet [86] (i.e., the



(a) Inference Latency (b) Power Consumption (c) Energy Consmption

Figure 4.2: State-of-the-art single-image HDR DNN models profilings.



(a) CDF (b) Overlap Ratio

Figure 4.3: Foveation study

most lightweight model among the tested DNNs), would deplete 20% of the battery. This clearly indicates the need for more efficient HDR content generation techniques tailored for edge devices.

4.2.2.2 Foveation in Human Visual System

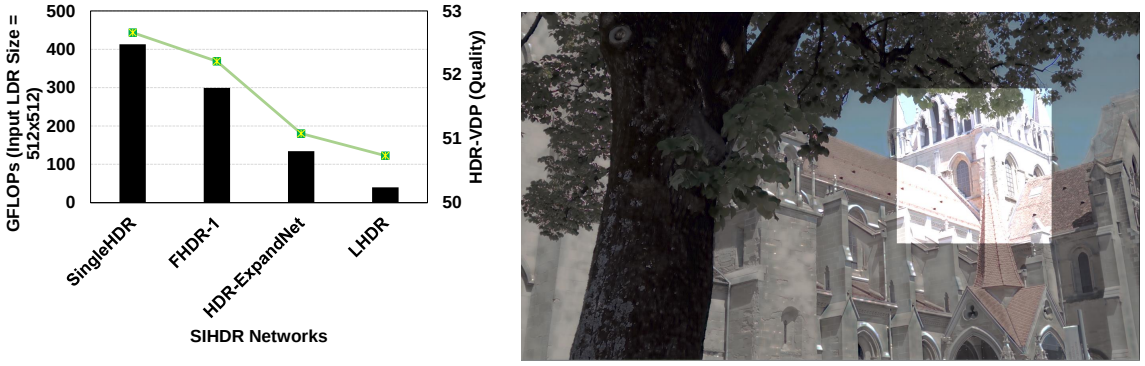
As discussed in Sec. 4.2.2.1, the high energy consumption of the state-of-the-art (SOTA) DNN models for SIHDR poses a significant challenge for their deployment on edge devices without additional optimizations. This motivates us to explore the techniques that can facilitate the efficient deployment of such models. Note that, while designing an efficient DNN model architecture from scratch could potentially address this inefficiency issue,

it requires the developers to have domain-specific knowledge of machine learning and also, substantial resources for training the DNN models. Therefore, our focus in this paper is on improving the execution efficiency of the existing DNN models. One potential approach for achieving this is to leverage the foveation feature of the human visual system (HVS), thereby reducing computation by allocating fewer resources to the regions outside the user's fovea. The intuition behind this approach stems from the fact that the spatial resolution of the HVS is highest around the foveation point and decreases rapidly with increasing eccentricity [**?**]. In fact, foveation-based optimization has been applied in prior works to improve the efficiency for various applications such as AR/VR, cloud gaming, xx []. In order to utilize this foveation feature for optimizing HDR content generation, we need to answer the following two critical questions regarding its validation and feasibility: *1). Will such foveated viewing phenomenon remain when users view HDR content? and 2). Given that a user's visual attention on HDR images is not available until the HDR images are reconstructed, how can we utilize this attention information to optimize the reconstruction process beforehand?*

To verify if such foveation feature remains when viewing HDR content, we analyzed the fixation density maps (FDMs) collected from users viewing HDR images from the HDREye dataset [] and draw the cumulative distribution function (CDF) envelopes for the fixation density value distributions across all FDMs in Fig. 4.3a. The density values range from 0 to 255, with a higher fixation density value corresponding to more focus []. As we can observe, over xx% of the fixation density values fall within 30 (which is quite small) for the orange curve, indicating that users primarily focus on a very small portion of the image (i.e., the regions with large fixation density values). Even for the lower envelope (yellow line), xx% of the fixation density values are within xx, meaning that a substantial area receives minimal attention. These observations confirm the presence of foveality with HDR visual content and validate the potential of foveation-based optimizations for efficient HDR content generation.

To answer the second question, we investigated the correlation between users' fovation on LDR images and the corresponding HDR versions in Fig. 4.3b. Recall that a higher fixation density value corresponds to more attention from the user, as such, we first define a threshold[1], the regions with fixation density values higher than this threshold are regarded as the foveated regions or regions of focus (RoFs). Following this, we can obtain the foveated regions for both the LDR and HDR images using their FDMs, respectively. Fig. 4.3b shows the overlap ratio (y-axis) between the foveated regions for LDR and HDR

[1]The threshold is set as p50 of the fixation density values of the given FDM.



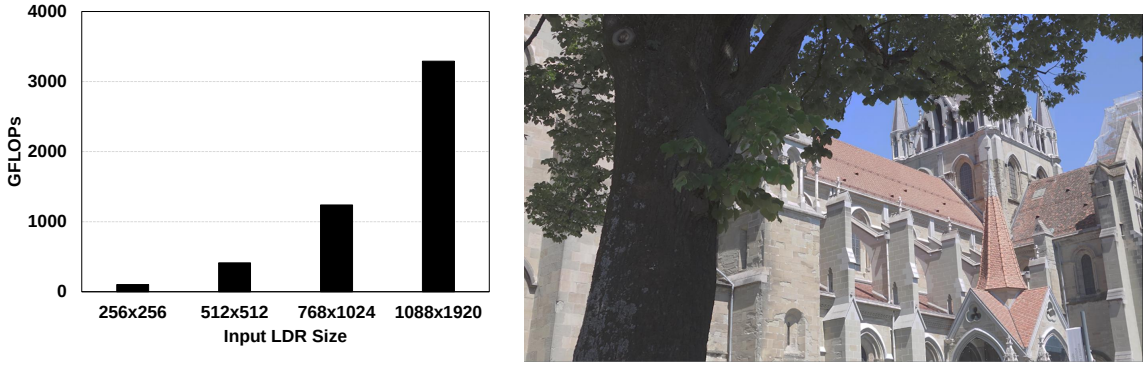
(a) SIHDR Tradeoff (b) Artifact

Figure 4.4: Opportunity-1: utilize the tradeoff between SIHDR models

images across different frames (x-axis). As can be observed from the figure, the overlap ratio is higher than 80% for most of the frames, suggesting that visual attention is likely to remain the same when users switch from LDR to HDR images. Upon observing this, it is straightforward, but promising, to utilize the collected gaze tracking information from LDR viewing to predict foveated regions for HDR images.

4.2.2.3 Opportunities

Sec. 4.2.2.2 presents the validation and feasibility of leveraging the foveation-based optimizations to improve the performance and energy efficiency of on-device single image HDR, however, after obtaining the RoF from the gaze tracking data, how to better utilize such RoF information to maximize the performance improvement while preserving the quality of the generated HDR image is not trivial. While the prior studies [90] have shown that in general, the model accuracy is positively correlated with the model complexity, one may consider applying lighter (but less accurate) DNN models or even non-ML-based techniques for the non-RoFs, and heavier (but more accurate) models for the RoFs. However, this strategy is not feasible for HDR reconstruction since different SIHDR models will produce HDR images with varying luminance levels. Simply concatenating these HDR images together will result in visible artifacts. For example, as shown in Fig. 4.4b, where we generate the HDR for RoF using SingleHDR [85] (the most complex model with the highest FLOPs and yields the best HDR image quality as shown in Fig. 4.4a) and the HDR for background using LHDR [84] (the most lightweight model with the least FLOPs and moderate HDR image quality), the concatnated HDR image exhibits visible artifacts. Despite these challenges, we believe this direction is worth exploring further. In the future, we plan to jointly train two SIHDR DNN models, such



(a) Size vs GFLOPs (b) No artifact

Figure 4.5: Opportunity-2: utilize the tradeoff between size

that they can reconstruct HDR images with similar characteristics and one model will prioritize image quality (e.g., for RoI), while the other will focus more on performance and efficiency (e.g., for background regions).

An alternative approach to reducing the computational burden for background regions is to downscale the input resolution for these regions. Unlike the traditional classification or object detection models like YOLO [], MobileNet [], etc. which usually require a fixed input size, the SIHDR networks allow the variable input size, typically matching the size of the input LDR image. To understand the relationship between the size of the input LDR image and the computational demands, we vary the input size and plot the corresponding GFLOPs values for each input size for SingleHDR [85] in Fig. 4.5a. As we can observe, the computational complexity decreases dramatically as the input size reduces. Such observation motivates us to first downscale the LDR background region before feeding it to the SIHDR network and upscaling it back after the HDR content is generated. While the downscaling and upscaling can be achieved using conventional imaging processing techniques like bilinear interpolation [] or cubic interpolation [], the performance gains and energy savings are expected to be significant. Moreover, as shown in Fig. 4.5b, the artifact issue as we previously discussed can be avoided if we reconstruct both the RoI and background regions using the same SIHDR model, which demonstrates a promising avenue for optimizing HDR reconstruction on edge devices.

4.3 Design

We have discussed the potential opportunity of optimizing HDR image reconstruction by allocating less computation to background regions in Sec. 4.2.2, in this section, we

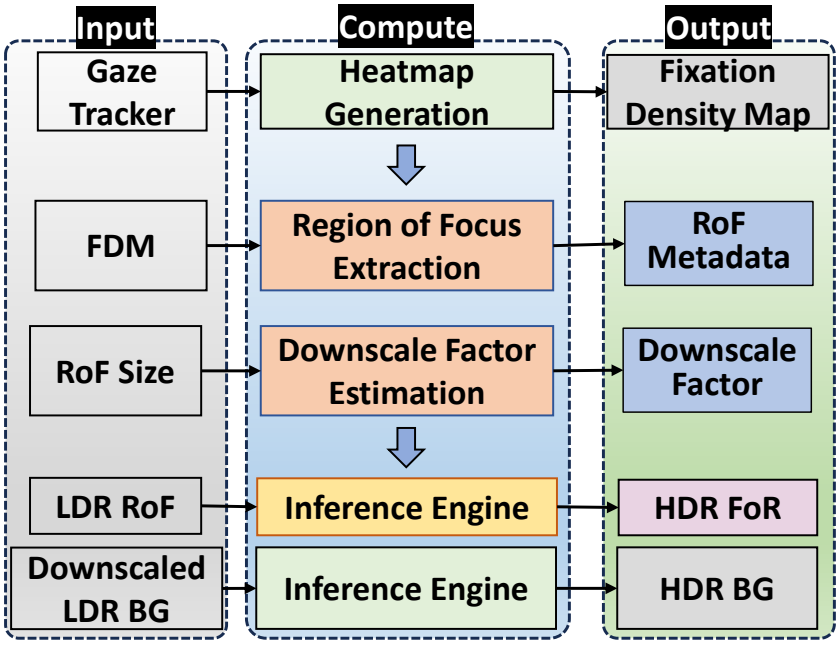


Figure 4.6: Foveated HDR overview.

will dive into the detailed designs for utilizing such opportunity. We begin by providing the overview of our design in Sec. 4.3.1, followed by a detailed explanation, including the determination of the region of focus (RoF) and the selection of the resolution or downscale factor for the background regions in Sec. 4.3.2. Finally, Sec. 4.3.3 addresses the issue of potential quality drops caused by such optimization/approximation to ensure the generated HDR image retains optimal quality.

4.3.1 Overview

Fig. 4.6 illustrates the overview the our proposed Foveated HDR, which primarily consists of three steps, including: 1). Fixation density map (FDM) generation using the collected gaze tracking data from the sensor; 2). Leveraging the generated FDM to determine the Region of Focus (RoF) as well as the downscaling factor for the background regions; 3). Downscaling the background accordingly and performing inference for the RoF and the downscaled background.

Fixation Density Map (FDM) Generation is a relatively well-established field with extensive exploration [**?**]. Specifically, with the eye tracking sensor, the FDM can be generated using the collected gaze pattern. Moreover, several open-sourced implementations for FDM generation exist, such as []. Given the maturity of this field, we will not dive deep into the details for the FDM generation algorithms and do not

claim this as our contribution. Instead, we assume that the FDM has been pre-generated for the next steps (i.e., the contributions of this work).

RoF Determination and Downscaling Factor Selection: Given the obtained fixation density map, we then employ Algo. 3 (discussed in detail later) to extract the metadata for the region of focus (RoF), which includes its center coordinates, as well as the size/dimension. After that, we can leverage this information to estimate the potential inference latency for the RoF, and then select a proper downscaling factor for the background region in conjunction with the user's desired speedup target. The selected background downscaling factor aims to ensure that the total reconstruction time for both RoF and background regions meets, or closely approximates, the user's speedup expectation.

RoF-assisted HDR Reconstruction: The extracted RoF metadata and background downscaling factor allow us to isolate the low dynamic range (LDR) RoF and low-resolution LDR background regions. While simply concatenating the reconstructed HDR RoF and background may appear feasible (as discussed in Section 6.3.2), this approach may not always yield optimal results. To ensure high-fidelity HDR images, we need to carefully consider the DNN model used for HDR reconstruction and make necessary modifications to ensure the generated HDR image retains optimal quality, as we will elaborate more later in Sec. 4.3.3.

4.3.2 Extracting Region of Interest and Adjusting Background Resolution

4.3.2.1 Extract Region of Focus

As we have discussed in Sec. **??**, higher fixation density values in the FDM indicate more focus from the user. Driven by this observation, we propose a two-step algorithm (Algo. 3) for determining the center coordinates and size for the region of focus (RoF), respectively. The first loop (line#3 - #9) aims to determine the center coordinates for the region of focus (RoF) by traversing the input fixation density map $M_H \times W$, where H and W denote the height and width of the FDM, respectively (matching those of the input low dynamic range image). Specifically, in each iteration, we first extract a region of size $D_L \times D_L$ from the input FDM and calculate its density sum (line#5 - #6), where D_L is the pre-defined lower bound for the RoF size. If this sum exceeds the maximum density sum obtained so far, we will update the maximum density sum (max_sum) as well as the RoF center coordinates ($center$) accordingly (line #7 - #9).

After obtaining the RoF center coordinates in the first loop, the average fixation density value within the current RoF region of size $D_L \times D_L$ is calculated (line#10). The RoF size is then iteratively expanded outward from its initial size D_L. As shown in line#12 in Algo. 3, in each iteration, we increase the RoF size by the fixed step size (i.e., S), and extract the corresponding sub-region from the FDM (line#13 - #17). The average fixation density value for the expanded area is then calculated (line#18 - #19). If this average value falls below a threshold compared to the average obtained previously (line #10), this means that the current RoF size is sufficient and has already included regions with smaller fixation density values, which may not be of interest to the users. Therefore, the expansion process is stopped and the corresponding RoF size is returned (line#20 - #21). Otherwise, the RoF continues to expand until reaching the pre-defined upper bound.

4.3.2.2 Determine Background Downscale Factor

As depicted in Sec. 6.3.2, reducing the resolution of the background region presents an opportunity to save the computation and energy consumption for HDR reconstruction. However, the optimal downscaling factor for the background remains unknown and requires careful selection. While a larger downscaling factor can preserve the reconstructed HDR image quality at the cost of decreased performance improvement, a smaller factor can lead to significant speedups but potentially at the expense of quality. Rather than randomly selecting a downscaling factor without any prior knowledge, we propose a user-driven design and allow the user to determine this factor. It is important to note that, we do not expect users to provide the exact downscaling factor, which would be user-unfriendly. Instead, the only required parameter from user-end is their desired speedup[2] by employing our Foveated HDR proposal. Based on this target speedup as well as the estimated execution latency required for inferencing the RoF, we can predict the downscaling factor for the background region such that the desired speedup can be achieved. Towards this, a critical question arises – *how do we estimate the execution latency for RoF with only its center coordinates and size given?* To answer this question, we analyze the relationship between RoF size and the inference latency through profiling as shown in Fig. 4.7. Specifically, the left y-axis shows the required GFLOPs while the right y-axis shows the inference latency on Jetson Orin Nano edge board [88], with

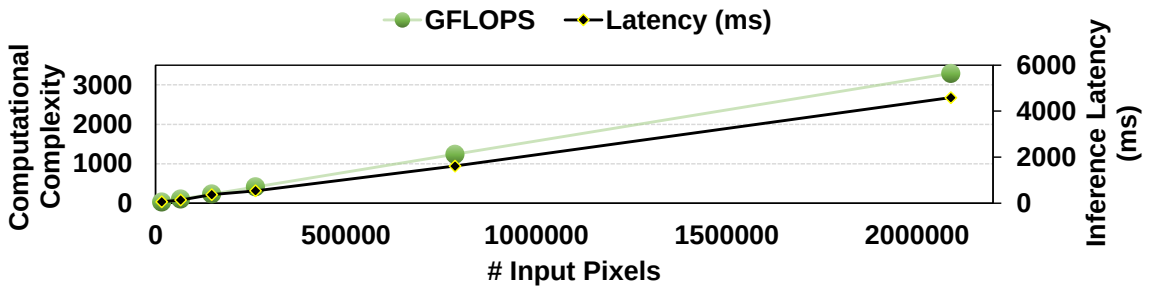
[2]We are referring to the speedup for the forward pass computation during DNN inference, while there are extra overheads for searching for RoF and downscaling/upscaling the background region included in our proposed Foveated HDR, the final end-to-end HDR content generation speedup is expected to be lower than the target speedup specified by the user.

Algorithm 3: Computation of Region of Interest Using Fixation Density Map

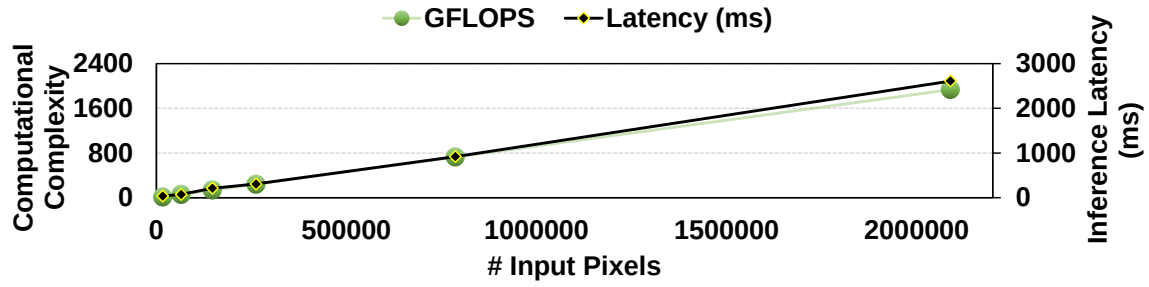
Input : $M_{H \times W}$: Fixation density map
Input : $(D_L,\ D_U)$: Lower and upper bounds for the RoI dimension
Input : S: Step size for RoI expansion
Input : T: Threshold for terminating RoI expansion
Output : $center$=$(h,\ w)$: Center coordinates of the RoI
Output : d: Dimension of the RoI

Procedure `ComputeRoI`($M_{H \times W}$, D_L, D_U, S, T)
  Init: $max\_sum = 0$, $center = (0, 0)$, $d = D_L$
  for $h \leftarrow 0$ **to** H **by** D_L **do**
    for $w \leftarrow 0$ **to** W **by** D_L **do**
      $roi = M[h : h + D_L,\ w : w + D_L]$
      $density\_sum = sum(roi)$
      if $density\_sum > max\_sum$ **then**
        $max\_sum = density\_sum$
        $center = (h + D_L/2, w + D_L/2)$
  $avg_0 = max\_sum / D_L^2$
  while $d < D_U$ **do**
    $d = d + S$
    $start\_h = max(0,\ center[0] - d/2)$
    $start\_w = max(0,\ center[1] - d/2)$
    $end\_h = min(H,\ center[0] + d/2)$
    $end\_w = min(W,\ center[1] + d/2)$
    $roi = M[start\_h : end\_h,\ start\_w : end\_w]$
    $density\_sum = sum(roi)$
    $avg = (density\_sum - max\_sum)\ /\ (d^2 - D_L^2)$
    if $avg/avg_0 < T$ **then**
      return $center$, d
  return $center$, d

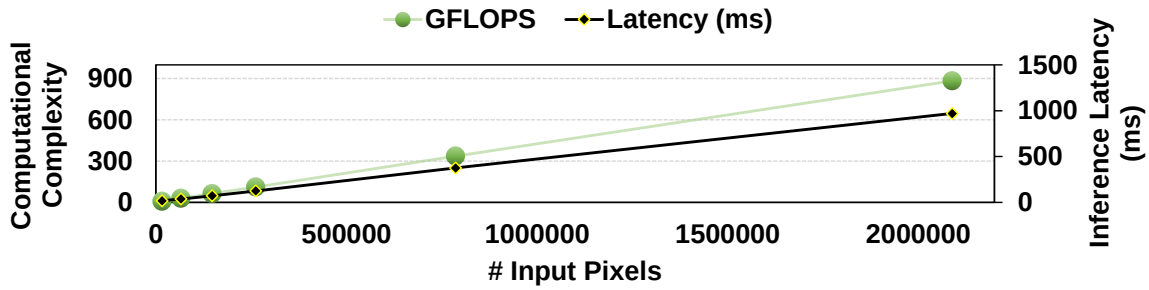
various input LDR image sizes as listed in Table. 4.1. As we can observe, for all of the three tested SIHDR DNN models, both the computational complexity (i.e., GFLOPs) and the execution latency increase *linearly* along with the total number of pixels of the input LDR image. This observation motivates us to use the following equation (Eq. 4.1) to estimate the execution latency for the RoF, where the S_{RoF} and $S_{Input\ Image}$ denote the area for the input RoF and the original image, respectively, and are equivalent to their total number of pixels.



(a) SingleHDR



(b) HDRCNN



(c) HDR-ExpandNet

Figure 4.7: Computational complexity and inference latency for LDR images with varying sizes.

$$T_{RoF} = \frac{S_{RoF}}{S_{Input\ \ Image}} * T_{Input\ \ Image} \tag{4.1}$$

Similarly, we can estimate the inference latency for reconstructing the downscaled background using Eq. 4.2:

$$\begin{aligned} T_{Background} &= \frac{S_{Background}}{S_{Input\ \ Image}} * T_{Input\ \ Image} \\ &= factor^2 * T_{Input\ \ Image} \end{aligned} \tag{4.2}$$

Table 4.1: Tested input low dynamic range (LDR) image size and the number of pixels.

LDR Size	256×256	384×384	512×512	768×768	1088×1920
# Pixels	65536	147456	262144	786432	2088960

where $factor$ is the downscaling factor for the background region, and ranges from 0 to 1. Combining the definition for inference speedup (Eq. 4.3) with Eq. 4.1 and 4.2, we can derive the downsampling factor for the background region using Eq. 4.4.

$$\frac{T_{Input\ Image}}{T_{RoI} + T_{Background}} = speedup \tag{4.3}$$

$$factor = \sqrt{\frac{1}{speedup} - \frac{S_{RoI}}{S_{Input\ Image}}} \tag{4.4}$$

4.3.3 How to Preserve Quality?

As we discussed in Sec. 6.3.2, applying different SIHDR DNN models for the RoF and background region will lead to visible artifacts on the generated HDR image. To overcome this issue, we propose applying the same SIHDR DNN model to both RoF and background regions. While this approach is effective in most cases (as we will show in Sec. 4.4.3.4), sometimes it may not work. For instance, as shown in Fig. **??**, where both the RoF and background are inferenced using the HDR-ExpandNet [] DNN model while the background is downscaled by a factor of 0.5 (RoF remains at original resolution). This will save xx% inference computation, however, as shown in the figure, visible artifacts appear around the edges where the RoF and background connect. To understand the reason causing these artifacts, we delve into the detailed architecture of HDR-ExpandNet []. As shown in Figure **??**, HDR-ExpandNet consists of global, local, and post-processing modules. The global module considers the entire image, while the local module focuses on generating local details. Simply feeding the RoF to HDR-ExpandNet allows it to treat the RoF as the global image. When the visual characteristics (e.g., luminance) between the RoF and the whole input LDR are significant, the generated HDR RoF (based on a "fake" global image, i.e., the RoF itself) may not be compatible with the HDR generated for the whole image. This incompatibility leads to visible artifacts at the edges. To address this issue, we propose to skip the global module computation for the RoF inference. Instead, we reuse the output of the global module obtained during background inference. This will ensure that the global image information is captured and utilized when processing the RoF, thereby eliminating the artifacts. As shown in Figure **??**, the

proposed modifications towards the HDR-ExpandNet [] effectively removes the artifacts observed in Figure **??**. To summarize, while applying the same SIHDR DNN model to both RoF and background regions offers an effective solution to eliminate artifacts, we also need to carefully consider the SIHDR model architecture, especially, when handling models like HDR-ExpaneNet which utilizes the global context for HDR construction.

4.4 Evaluation

In this section, we compare our proposed Foveated HDR design against the baseline setting where all of the HDR frames are generated with equal computational resources allocated to all pixels. We evaluate four critical metrics relevant to HDR content generation applications: execution latency, power and energy consumption, as well as the quality of the generated HDR images. Towards this, we first describe the experimental platform, datasets (Sec. **??**), and configurations (Sec. **??**), and then analyze the collected results in Sec. **??**.

4.4.1 Methodology

4.4.1.1 Experimental Platform

To evaluate the effectiveness of our proposed Foveated HDR, we use the NVIDIA Jetson Orin Nano [] board as the experimental platform, which is an edge development board, and is well-known to simulate the realistic edge development environment. Specifically, it is equipped with a 1024-core NVIDIA Ampere GPU with 32 Tensor Cores, a 6-core Arm Cortex-A78AE v8.2 64-bit CPU, 8GB 128-bit LPDDR5 memory and uses a 256GB MicroSD card as the storage. In our evalutaion, we config the Jetson Orin Nano board to operate in 15W compute mode and monitor its power consumption using the tegrastats tool [].

4.4.1.2 HDR Dataset

We employ two datasets, namely HDR-Eye [] and DML-iTrack-HDR [], for our evalutation. The HDR-Eye is an image dataset consists of 46 HDR images and their corresponding LDR version. Each of the LDR and HDR images has its corresponding fixation density map (FDM) showing the users' view of focus when viewing the HDR/LDR images. [3] The

[3]Only 36 of the LDR/HDR image pairs are considered as valid for evaluation since they contain all the necessary componenets such as the LDR images, HDR images, and FDM or fixation points for the

Table 4.2: Dataset

Name	HDR-Eye	Bistro_01	Carousel	Fishing	Playground
#Frames	36	151	339	371	222
Resolution	1920×1080	1920×1080	1920×1080	1920×1080	2048×1080

DLM-iTrack-HDR dataset [] contains several HDR videos and the corresponding fixation density maps showing the user's gaze tracking for each frame when viewing the HDR videos. However, it does not include the corresponding LDR videos or the FDMs for the LDR videos. To address this issue, we manually generated LDR versions of the HDR videos using a tone mapper, as commonly done in prior works like []. Furthermore, since our previous results in Sec. **??** indicate large overlaps between the focus areas when users watch LDR and HDR contents, we decide to use the fixation density maps collected on the HDR videos to simulate the FDMs for the LDR inputs. Table. **??** summarizes the details for each dataset and videos used in this evaluation, including the total number of frames and the resolution.

4.4.1.3 SIHDR DNN Models

To demonstrate the generalizability of our proposed foveated HDR design, we evaluated it on three widely-used SIHDR DNN models with varying characteristics and computational demands: SingleHDR [], HDRCNN [], and HDR-ExpandNet []. The SingleHDR [] and HDRCNN [] are implemented using TensorFlow v1 [], while HDR-ExpaneNet [] utilizes the Pytorch vxx []. We leverage the pretrained weights provided by the original authors for all models.

4.4.2 Design Configurations

We evaluate our proposal against the baseline setting under two different configurations:
Baseline: We implement the baseline HDR content generation on an edge GPU board (NVIDIA Jetson Orin Nano []), where every HDR frame is generated using the SIHDR DNN inference and all the pixels are treated equally, regrdless of their importance to the user's visual attention.
Speedup_2X: As discussed in Sec. 4.3.2.2, Foveated HDR is user-driven, where the resolution for background region is adaptively decided based on the user-specified speedup.

LDRs.

Specifically, after determining the computational resources required for the regoin of focus (RoF) using the algorithm described in Sec. 4.3.2.1, the background is downscaled by a factor before performing the inference, so that we can save the computation and achieve the target speedup specified by the user. We first set the target speedup as 2, which aims to provide a moderate speedup with minimal quality loss.

Speedup_4X: In this configuation, we set a larger target speedup to achieve greater latency savings and energy consumption reduction. This setting prioritizes reducing latency and energy consumption over the HDR content quality and is suitable for applications where content is consumed quickly, such as fast playback of videos, and users are less sensitive to quality changes in the background due to their focus on the RoF. For example, mobile devices with limited battery budgets may benefit from this configuration , as it saves significant energy while maintaining acceptable visual quality for the users' attention focus.

4.4.3 Results

We compare the execution latency, power and energy consumption, as well as the HDR content quality in Fig. 4.8 - Fig. 4.10 and Table. 4.3, when using various design configurations as described in Sec. 4.4.2.

4.4.3.1 Execution Latency

Latency for Inference Computation: We first compare the inference latency consumed by the SIHDR models during forward pass computation under difference configurations. Specifically, as shown in Fig. 4.8a, the baseline SingleHDR [] model requires an average of 4.66s to generated one HDR frame, while with our proposed Foveated HDR optimization, the inference latency can be decreased to 2.41s and 1.34s per HDR frame for Speedup_2X and Speedup_4X configurations, respectively. These reductions translate to speedups of 1.93x and 3.48x compared to the baseline, respectively. Note that, these achieved speedups are slightly lower than the user-specified speedups (i.e., 2x and 4x). This discrepancy arises because the input LDR image needs padding to ensure its dimensions are a multiple of 64 before being fed into the SingleHDR model [], which will add additional computations. For the Speedup_2 configuration, the moderate target speedup allows the background region to remain relatively large. Therefore, the additional computational cost introduced by inferencing the padded regions has a

minimal impact on the inference latency, yielding the real speedup closely matches the expected speedup estimated using Equ. **??** in Sec. 4.3.2.2. However, for the Speedup_4X configuration, the significant background downscaling leads to a substantial reduction in total computation. As a result, the extra computation introduced by inferencing the padded background region becomes non-negligible, since it now accounts for a larger portion of the overall computation. This explains why the real speedup is lower than the configured speedup. Similarly, as shown in Fig. 4.9a, the inference latency for HDRCNN [] can be reduced from 2.66s/frame to 1.34s/frame (1.985x speedup) for Speedup_2X and 0.73s/frame (3.64x speedup) for Speedup_4X configuration, respectively. While for HDR-ExpandNet [] (Fig. 4.10a), our proposed Foveated HDR is able to reduce the inference latency for generating one HDR frame from 1.04s in baseline setting to 0.54s and 0.28s for Speedup_2X and Speed_4X configurations, respectively, corresponding to speedups of 1.93x and 3.71x.

Latency for End-to-end HDR Generation: Fig. 4.8b, Fig. 4.9b and Fig. 4.10b compare the end-to-end execution time for generating one HDR frame using the baseline configuration and our proposed Foveated HDR design. While our approach introduces additional overhead due to the Region-of-Focus (RoF) search, LDR background downscaling, and HDR background upscaling (shown in the yellow bars), this overhead is minimal compared to the performance gains achieved. Specifically, the Foveated HDR overhead accounts for approximately 1.6%, 3.0% and 8.2% of the baseline end-to-end inference latency for SingleHDR [], HDRCNN [], and HDR-ExpandNet [] models, respectively.

4.4.3.2 Power Consumption

To study the impact of our proposal in terms of power consumption, we measure the power consumed during HDR content generation for different configurations in Fig. 4.8c, Fig. 4.9c and Fig. 4.10c. As we can observe, compared with the baseline inference, the Foveated HDR can achieve 10%, 9% and 18.5% power saving for SingleHDR [], HDRCNN [] and HDR-ExpandNet [], respectively, under the Speedup_2X configuration. These power savings primarily stem from two aspects: 1. the amount of computation allocated for the background region is reduced, which leads to lower power consumption by the compute units (e.g., CPU/GPU); and 2). downscaling the LDR background translates to a smaller input size for the SIHDR model, which reduces data movement and memory consumption and leads to further power savings in memory operations. Moreover, with the more aggressive downscaling employed by the Speedup_4X configuration, we can achieve an additional 7% - 10% reduction in power consumption.

Table 4.3: Quality

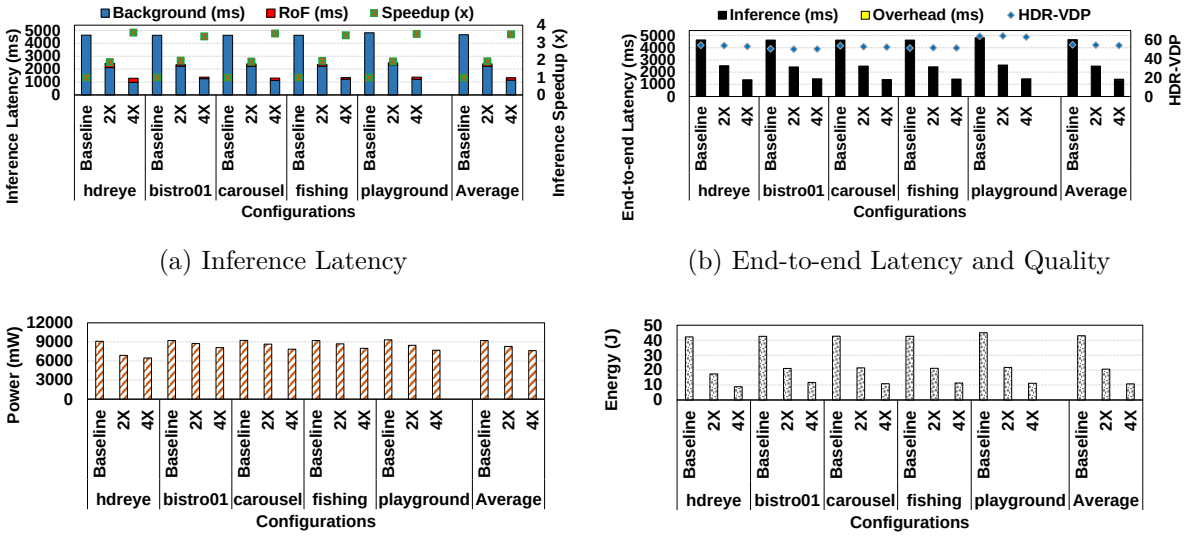
		hdreye			bistro_01			carousel			fishing			playground		
SIHDR Model	Config.	PNSR	SSIM	LPIPS	PNSR	SSIM	LPIPS	PNSR	SSIM	LPIPS	PNSR	SSIM	LPIPS	PNSR	SSIM	LPIPS
SingleHDR	Baseline	**23.09**	**0.82**	**0.16**	**18.23**	**0.78**	**0.35**	**19.54**	**0.80**	**0.18**	19.55	**0.87**	**0.22**	**23.47**	**0.92**	**0.08**
	2X	22.71	0.81	0.20	17.14	0.76	0.38	18.56	0.77	0.23	20.00	**0.87**	0.24	23.38	**0.92**	**0.08**
	4X	22.45	0.80	0.27	16.82	0.75	0.40	17.63	0.73	0.29	**20.93**	0.86	0.28	23.17	0.90	0.11
HDRCNN	Baseline	18.30	**0.70**	**0.25**	**29.41**	**0.85**	**0.30**	27.32	0.86	**0.11**	**31.50**	**0.93**	**0.18**	27.60	**0.95**	0.05
	2X	**18.37**	**0.70**	0.26	27.31	**0.85**	0.32	27.53	**0.87**	**0.11**	30.66	0.92	0.20	27.32	**0.95**	**0.04**
	4X	18.36	0.69	0.31	28.03	**0.85**	0.35	**27.54**	0.86	0.13	30.72	0.92	0.23	**27.75**	0.94	0.06
HDR-ExpandNet	Baseline	**21.41**	**0.79**	**0.21**	**21.28**	**0.81**	**0.35**	**22.57**	**0.84**	**0.16**	**23.66**	**0.89**	**0.23**	**26.72**	**0.93**	**0.07**
	2X	21.24	**0.79**	0.22	20.93	0.80	0.36	22.36	**0.84**	0.17	23.46	**0.89**	0.26	26.49	**0.93**	**0.07**
	4X	20.91	0.78	0.28	20.98	0.80	0.38	22.19	0.83	0.20	23.34	0.88	0.28	25.54	0.92	0.08

4.4.3.3 Energy Savings

As shown in Fig. 4.8d, compared to the baseline, the Foveated HDR generation approach only consumes xx% energy on average for the SingleHDR [] model, under the Speedup_2X configuration. The energy saving can be further increased to xx% with the aggressive downscaling of background region in Speedup_4X configuration. Similar trends can also be observed for the HDRCNN [] and HDR-ExpandNet [] models in Fig. 4.9d and Fig. 4.10d, where the Speedup_2X configuration can yield xx% and xx% energy saving, while Speedup_4X can deliver xx% and xx% energy saving for HDRCNN [] and HDR-ExpandNet [], respectively.

4.4.3.4 Quality

1. hdrvdp 2. ldr quality 3. modified the hdr-expandnet



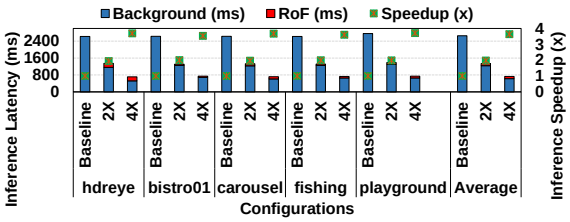
(a) Inference Latency

(b) End-to-end Latency and Quality

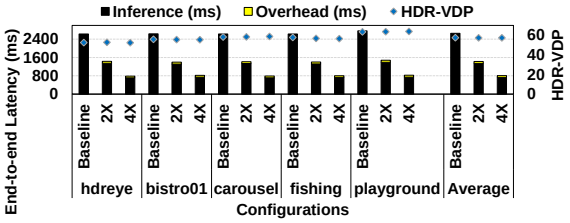
(c) Power Consumption

(d) Energy Consumption

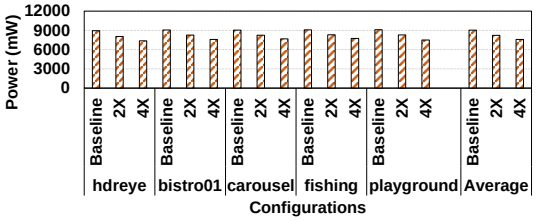
Figure 4.8: SingleHDR



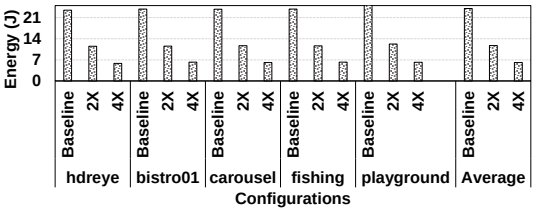
(a) Inference Latency



(b) End-to-end Latency and Quality

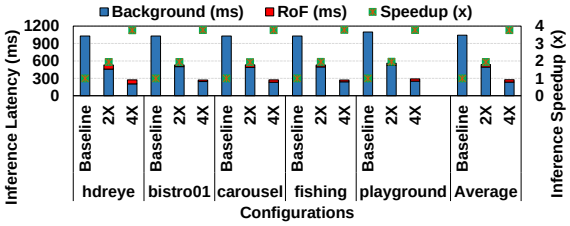


(c) Power Consumption

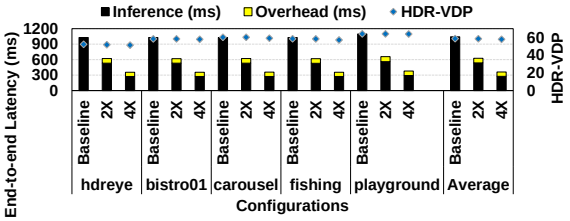


(d) Energy Consumption

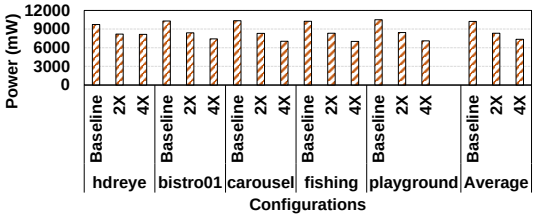
Figure 4.9: HDRCNN



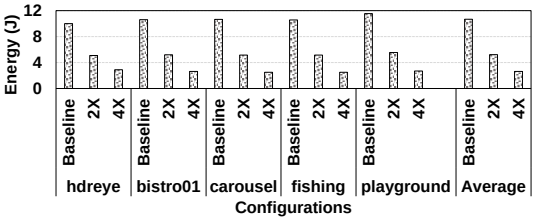
(a) Inference Latency



(b) End-to-end Latency and Quality



(c) Power Consumption



(d) Energy Consumption

Figure 4.10: HDR-ExpandNet

Chapter 5
Pushing Point Cloud Compression to the Edge

5.1 Introduction

As the world is increasingly becoming virtual and moving closer towards automation, accurate 3D representation of real-life objects in the virtual domain, be it for life-like graphics or efficient autonomous driving, is becoming essential. Recently, *Point Cloud* (PC) consisting of millions of points, which capture the 3D geometry and attributes (e.g. RGB colors), has become an important modality for such realistic representations for applications like AR/VR, gaming, autonomous driving, etc. Moreover, with the recent pandemic, as telepresence is becoming a norm, people are virtually attending meetings, visiting arts, heritage sites and tourist places across the globe, and even living in a virtual universe. All these applications rely on high quality PC capturing, processing and displaying for a more realistic experience. Additionally, with the new generation mobile phones, capable of capturing PC and then streaming them into an AR/VR enabled head mounted displays (HMDs), capturing 3D PC now is becoming as common as capturing a photograph. With this trend, the PC business is expected to reach a 10 Billion dollar industry by 2024 [91].

Since capturing PC no longer requires sophisticated, commercial and expensive devices and people are equipped with mobile PC capturing devices (like iPhone 13 Pro [92]), the application providers are also pushing many of the PC processing tasks like compression or rendering to the edge, to avoid the use of expensive cloud resources, minimize the data transfer latency, and/or protect user's privacy. Considering the dense features, 3D geometry and the visual attributes captured in PC, especially for the media

applications like telepresence and virtual visits, pre-processing [93–96], compressing and storing [97–102], post-processing and streaming [103–107] PC using a mobile device, while maintaining a reasonable quality of service (QoS), are fast becoming challenging tasks. Specifically, *PC compression* (*PCC*, or *PC encoding*) consists of both geometry (e.g., x, y, z coordinates in the 3D space) and attribute (e.g., RGB colors) compression. Our experiments show that PCC is the most expensive computation in a PC processing pipeline that takes $\approx 4seconds$, especially when deployed in mobile/edge devices, and hence, is a major contributor to the performance, video quality, and transmission energy, for the entire PC pipeline.

However, it is challenging to design an optimal point cloud compression (PCC) pipeline which is *fast* (within or close to real-time), *accurate* (with good quality), and *efficient* (with high compression ratio). The state-of-the-art PCC pipeline typically utilizes tree structures like *Octree* [108] or *kd-tree* [109] for compression, and often, the tree construction becomes a bottleneck due to lack of parallelization. Moreover, the conventional PC typically stores the geometry, while a wide array of applications, especially the ones meant for content consumption, infotainment and gaming, need the attributes to be stored as well, hence making the compression even more complex. For example, TMC13 [110] and CWIPC [102] – two state-of-the-art (SOTA) PCC techniques – take $4.1s$ and $4.2s$, respectively, to compress one PC frame on an edge platform, which are significantly higher than the real-time requirement ($\approx 100ms$ [97]), making them even more challenging to employ in emerging edge devices.

To address this, we study the SOTA compression pipelines and observe that the main reason behind their performance inefficiencies is their *sequential updates* to the global result *with each intermediate local runtime state* in a *point-by-point* fashion. Moreover, there has been little effort in parallelizing them on the state-of-the-art commercial systems, let alone on any edge/mobile devices. Prior works on PCC acceleration [97, 111] only consider the PC with geometry data and/or have limited parallelism, and thus, could neither leverage GPU nor benefit from other types of accelerators. In this context, this paper explores the following three **opportunities**: ① The points can be processed *in parallel* by using Morton codes [112] (which mathematically represent the geometry relationship among points) to identify the spatial-locality[1] within one frame for geometry compression. ② Further, this locality also exists in attributes (RGB pixels), i.e., spatial locality leads to attribute similarities, and hence opening opportunities for fast attribute compression. ③ And, finally, the locality extends beyond a single frame, i.e., the temporal

[1]In this paper, we use "locality" and "frame similarity" alternatively.

locality, which can be leveraged by sorting the points in the Morton code order, creating further opportunities to improve the compression efficiency.

Motivated by these opportunities, we propose and evaluate a two-pronged compression approach, where the *intra-frame* approach leverages the opportunities described in ① and ②, and the *inter-frame* approach takes advantage of ③. The intra-frame approach speeds up the geometry and attribute compression by 37× and 49× respectively, while the inter-frame approach further improves the compression ratio by ≈ 1.75× by reusing the matched blocks in reference frame.

To the best of our knowledge, this is the first work that targets to push the PCC to the edge by taking edge device-specific constraints into account and targeting four critical metrics – latency, energy, quality, and compression ratio. The major **contributions** of this work are the following:

- We identify the spatio-temporal redundancies for optimizing the PCC using a public PC dataset [8]. We also demonstrate that, such spatio-temporal localities can be precisely captured by Morton codes [112]. Specifically, we find that ① the points with similar Morton codes within one frame tend to have little variances in both geometry and attribute values (*spatial locality*), and ② the points with adjacent Morton codes (for instance, a cluster of geometrically close points) are likely to move in a certain direction, as a whole block, across frames (*temporal locality*).
- We propose two complementary designs to capture and utilize such spatio-temporal localities. First, we propose a Morton code-assisted intra-frame compression scheme, where *both* the geometry and attribute can be compressed in a *highly parallel fashion.* We believe this is the first work that applies the Morton code-based parallel octree construction algorithm [113] to speed up PC geometry compression. On the other hand, for attribute compression, we propose to sort the points in the Morton code order with the goal of capturing the attribute similarities. Also, to utilize temporal locality, we propose an inter-frame compression scheme which further increases the compression efficiency.
- We implement and evaluate our proposals on an edge device – NVIDIA Jetson AGX Xavier board [17]. Our extensive experimental results show that, compared to a state-of-the-art intra-frame PCC technique [110], our intra-frame proposal can accelerate the PCC by 43.7× and save 96.6% energy. While with our inter-frame compression design, the compression ratio can be further improved (increasing from 5.95 in intra-frame design to 10.43) with 35× speedup and 97.4% energy savings with respect to a state-of-the-art inter-frame PCC scheme [114]. Moreover, our proposal not only accelerates

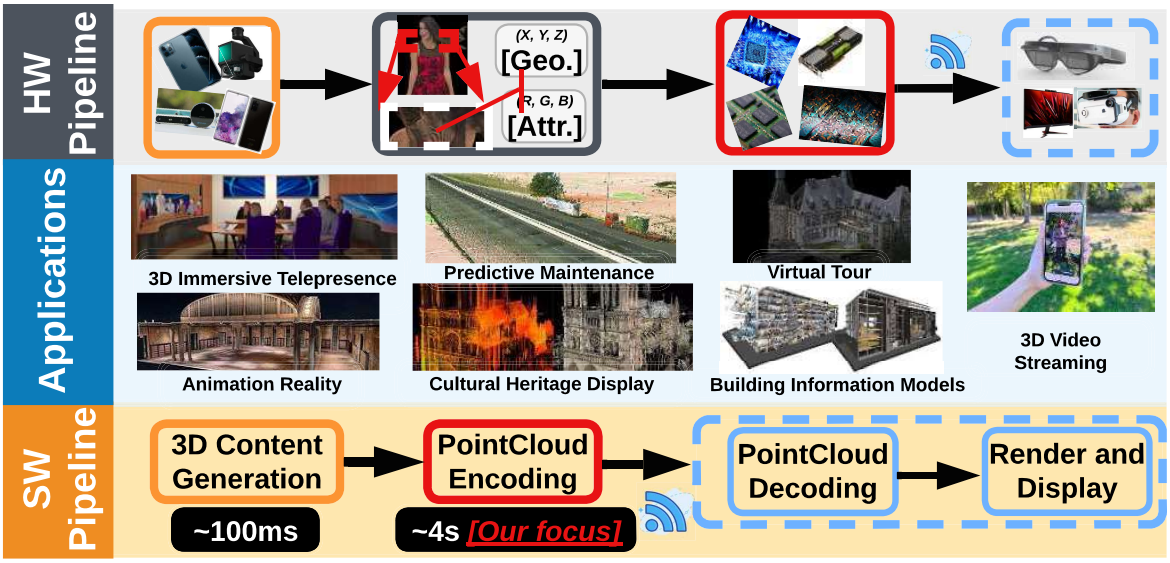


Figure 5.1: Example PC applications and processing pipelines.

the PC encoding stage, but also can improve the performance of the decoding stage which involves inverse encoding operations (e.g., reduces decoding latency to $\approx 70ms$), thus enabling the end-to-end PC processing in near real-time (i.e., 10FPS).

5.2 Background and Related Work

5.2.1 Background

Point Cloud in Real Life: Point Cloud (PC) is a set of points which represent objects or shapes in a 3D space where each point/voxel (3D equivalent of a 2D pixel) contains its 3D location (x, y, z coordinates), as well as some attributes (e.g., colors, normal, etc.). Capturing PC representation of the real world typically requires millions of voxels, far more than the amount of pixels required for 2D images. While PCs containing only the 3D geometry data are commonly used in LiDAR-based 3D imaging for autonomous vehicles or robotics path planning, the lack of attributes nullify their usage for visual media consumption. Therefore, any PC application meant for visual media like immersive telepresence, telemedicine, video streaming etc., needs the attributes to be stored along with the 3D coordinates.

Since PC generation requires sophisticated instruments like LiDAR or 3D cameras, it

was typically done on server-class computers with high compute and storage capabilities. However, with the advent of the modern mobile devices, capturing 3D image and PC on these tiny and battery-backed devices is becoming increasingly common. For example, the recent iPhone 12/13 Pro features LiDAR camera for PC recording, and similarly, Samsung Galaxy S20+/S20 Ultra contains ToF (Time of Flight) camera for the same. This makes PC-based media recording a common commodity, rather than a sophisticated pipedream. Moreover, applications like Record3D [93] enable seamless PC media streaming from phone to a wearable, encouraging a perpetually increasing PC content generation and consumption. However, the sheer volume of the data captured in these PC applications coupled with the limited compute and storage capabilities of these handheld devices pose a challenge in high quality PC media capture, storage and consumption [115]. To understand these challenges, we analyze an end-to-end PC pipeline.

End-to-end Pipeline: The PC video processing pipeline, as shown in Fig. 6.1, typically consists of 5 stages: 3D content generation, PC encoding, data transmission, PC decoding, render and display. In the *3D Content Generation* stage, the capturing device (e.g., the iPhone) uses LiDAR scanning or photogrammetry for the PC data acquisition. LiDAR maps spatial relationships and shapes by measuring the time taken by signals to bounce off objects and return to the scanner, while photogrammetry takes many photos from different angles to capture the target's geometry [116]. This process typically takes 10s of milliseconds [115]. Further, each point in the PC is associated with 3 coordinates (x, y, z) for the geometry and 3 colors (R, G, B) for the attribute. Thus, to represent one point, $4byte \times 3 + 1byte \times 3 = 15bytes$ are needed ($4byte$s per coordinate and $1byte$ per color component). Thus, a typical PC frame containing 10^6 points [117] require $120M$ bits of data, which is impossible to transmit in real-time to the end-user's display, from both the latency and energy standpoints, considering a steady 30-60fps requirement. Therefore, the PC video frame is compressed in the *PC Encoding* stage, before being transmitted over the *network* to the end-user. The received frame is decoded in the *PC Decoding* stage and the decoded PC frame is forwarded to the *Render and Display* stage where it is finally rendered and displayed on the screen. Note that although the same method is followed and has been well established for streaming $2D$ or 360° videos, given that the PC data is much denser, compression becomes essential as well as the primary bottleneck, often taking several seconds to compress one PC frame [115](see Fig. 6.1ⓐ).

5.2.2 Related Work

5.2.2.1 Point Cloud Use-cases

Recently, PC is being widely used in various fields, such as AR/VR [118, 119], telepresence [120–122], virtual tourism [123, 124], teleoperation [125], telemedicine [126], video streaming [127, 128] and gaming [118, 129], etc. where both geometry and attributes are essential as the contents are consumed by people for infotainment purpose. Almost all of these applications can be categorized as interactive volumetric video streaming. On the other hand, for applications such as autonomous driving [130, 131], robotics [132], motion planning [133] or path planning [134], attributes like RGB info, at most times, are not necessary as the PC is used in the compute pipeline (by the machine) to extract features and make decisions.

Especially, **interactive volumetric video streaming** is starting to become mainstream, as edge devices (e.g., iPhones) facilitate recording and streaming the PC video which provide end-users with real-time 6-degrees of freedom (6-DoF) experiences. Streaming such PC videos in real-time involves capturing both the attributes and geometry data making it a challenging task even without user-object interaction. Towards this, Han et al. proposed the viewpoint-dependent PCC scheme (termed as "ViVo") which only sends the 3D tiles within user's field of view [127], thereby reducing the data volume. Such optimizations [128, 135, 136] are extremely important for applications like virtual tourism, video streaming etc. More complex optimizations are needed when *human-object* interactions and *human-object-sensors* interactions are involved (for applications like telepresence, telemedicine, virtual shopping and gaming) and need the help of PC data analytics to recognize/classify the interactable objects/scenes.

5.2.2.2 Point Cloud Analysis

To analyze objects/scenes in PCs, 3D convolutional neural networks (CNNs) have been widely used in techniques like 3D shape classification [107, 137–139], object detection [140–142], tracking [143, 144], or segmentation [107, 137, 139, 145]. While most prior works target accuracy, Mesorasi [146] improves the compute and memory efficiency of 3D CNNs using delayed-aggregation and software-hardware co-design, and PointAcc [104] proposes special mapping unit and memory management for optimizations. However, the huge data volume of the PC still remains a bottleneck in data movement and sharing, creating hurdles in high quality-low latency streaming/analysis. Therefore, there have been several works focusing on compressing the PC, as discussed next.

5.2.2.3 Point Cloud Compression (PCC)

The prior PCC works can be classified as follows:

G-PCC utilizes special structures like octree or kd-tree to represent and compress the geometry [100, 147, 148]. For example, with octree-based PCC, considering a PC is contained in a $D \times D \times D$ cube, the cube is recursively divided into 8 $D/2 \times D/2 \times D/2$ sub-cubes until D=1. The occupied/non-empty voxels in level$_n$ can be indicated by the *occupy* bits of its "parent voxel" (voxel in level$_{n-1}$). Each branch node in the octree stores 8 occupy bits, indicating the occupancy of its children/sub-cubes. The attribute compression in the G-PCC depends on the geometry. As a result, the attribute and geometry are compressed separately. There are 3 methods for attribute compression in the G-PCC – *RAHT* [99], *Predicting Transform* [149], and *Lifting Transform* [149]. The main idea behind RAHT is to use the attribute values in a lower octree level to predict the values in the upper level. In contrast, Predicting Transform and Lifting Transform are based on the hierarchical nearest-neighbor interpolation [150]. Apart from these methods that compress a static PC, there also exist several attempts at optimizing the compression for dynamic PCs by exploring the "temporal redundancy" across the PC video frames. For example, a macro block (a $S \times S \times S$ cube) based motion estimation and compensation is proposed in [101, 102, 151, 152], to further improve the compression efficiency.

V-PCC[2] targets compressing PC videos. Specifically, given a PC video stream, V-PCC first performs 3D to 2D projection on each frame [155–158], and then encodes these 2D projections via traditional 2D image codec. Both G-PCC and V-PCC are widely adopted in MPEG standard [159], and since our proposals begin with G-PCC, thus, is also compliant with the MPEG PCC standard.

NN-PCC[2] takes the raw PC as input, and feeds it into a pretrained 3D CNN, which outputs the compressed PC stream [160, 161]. Several recent efforts have been put into optimizing the 3D CNN to increase the compression ratio and/or decrease the number of parameters in the neural network model [162, 163]. However, based on the results reported in [104], even with a custom 3D-CNN accelerator, only 2.5× speedup could be achieved for 3D-CNN compared to edge GPU. Considering that NN-PCC can take thousands of seconds to compress one PC frame [153], such a huge gap between the long execution latency of NN-PCC and the real-time refresh requirement of vision applications

[2]Although V-PCC and NN-PCC have high compression efficiencies, they are compute-intensive [153, 154], and consequently, are not the best option for mobile devices and are not considered in this work. Besides, most of the NN-PCC only focus on compressing geometry data [153], thus, is not applicable for this paper's target (i.e., mainly for vision applications where the attributes are essential).

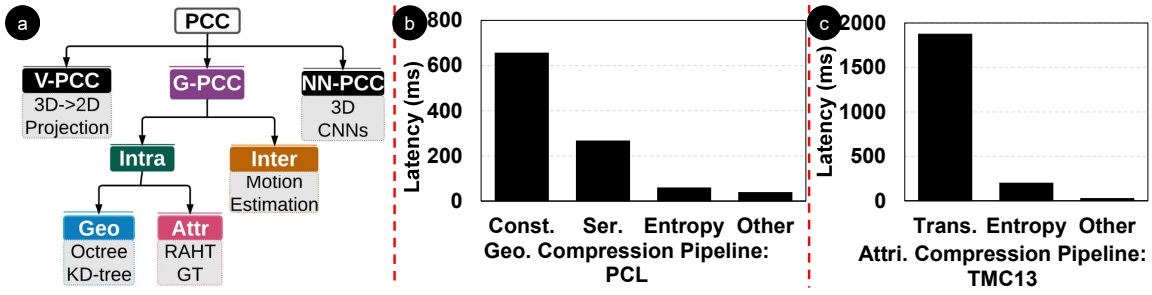


Figure 5.2: Prior PC compression technique categories and latency breakdown for prior techniques on compressing one PC frame from [3].

is yet to close and prevents the deployment of NN-PCC on edge devices. Moreover, NN-PCC mainly focuses on compressing the geometry data and hence is not very useful for the PC with attributes [153].

To the best of our knowledge, most of these works focusing on PCC with attributes target the compression ratio, and overlook the latency or energy consumption. However, as PC is moving to mobile, one cannot ignore the latency/energy constraints, thus demanding the need for mobile friendly PCC techniques which offer the best compression, latency and energy savings while preserving the video quality.

5.3 Motivation

5.3.1 Reasons for Inefficiency

To better understand the performance of the PCC pipeline, we characterize the "latency breakdown" of two state-of-the-art G-PCC techniques, i.e., PCL [164] and TMC13 [110], on a typical edge SoC platform (NVIDIA AGX Xavier) in Figs. 5.2ⓑ and ⓒ. Overall, the entire PCC pipeline takes around 3.5 seconds[3], which prevents one from employing such techniques in an edge device. Further, among the five stages in the pipeline, octree construction & serialization for geometry compression and RAHT for attribute compression are the two major bottlenecks which take $1s$ and $2s$, respectively. Driven by these observations, we investigate the reasons behind such inefficiencies, and

[3]We use PCL [164] and TMC13 [110] library for our profiling, where the geometry is compressed by the octree structure in PCL, and the attributes (RGB colors) are compressed through RAHT in TMC13.

further explore the potential opportunities for speeding up the PC compression.

Before delving into the details of our approach which aims to close the performance gap between "seconds" in practical and "hundreds of milliseconds" in ideal settings, we first investigate the reasons behind the inefficiencies of the prior techniques. Towards this, we studied three state-of-the-art PCC pipelines – octree-based pipeline for intra-frame geometry compression (Sec. 5.4.1.1), RAHT for intra-frame attribute compression (Sec. 5.4.3.1), and macro block-based motion compensation pipeline for inter-frame compression (Sec. 5.5) – and reached the following *conclusions*: the primary reason behind their performance inefficiencies is what can be termed as "under-parallelism", i.e., not being able to fully exploit parallelism during compression. Especially, many levels of dependencies (i.e, various regularities of locks) exist in their pipelines – e.g., the entire octree needs to acquire a "macro lock" before inserting a point and updating the tree (as shown in Fig. 5.5), during the intra-frame geometry compression; similarly, in the attribute compression shown in Fig. 5.6, the points at one layer in the octree have dependencies with those at other layers, thus their processing requires acquiring locks at a "layer granularity". Even with the optimizations in [111], where the octree construction stage can be performed in parallel, there are still several synchronization points, resulting to limited parallelism. To summarize, the performance inefficiencies in prior works can be primarily attributed to the lack of parallelism of these algorithms. Motivated by this observation, we next plan to improve the compression performance by exploiting various parallelism opportunities, which have been ignored, to the best of our knowledge, by the prior research but are essential in employing PCC in edge device settings.

5.3.2 What are the potential opportunities?

Increasing Geometry Compression Parallelism Using Morton Code: As mentioned earlier, the reason why the "sequential update" is necessary is that, during the intermediate stages, the *global* Octree (the final tree constructed at the last step) is unknown until the last point is inserted in the tree. To relax this constraint, if the PCs can be *sorted* based on a geometrical order, then the topographic structure of the global tree can be known at the beginning, thus fixing the tree structure and not requiring to be updated in a point-by-point fashion. As a result, these points can processed in *parallel*. In fact, there is a mathematical concept called *Morton Code* [112] (essentially, a space filling curve that maps a multidimensional data to one dimension while preserving the locality of the data points), which describes the geometrical location relationships between points, and thus can serve this purpose perfectly. There have been prior works like N-body

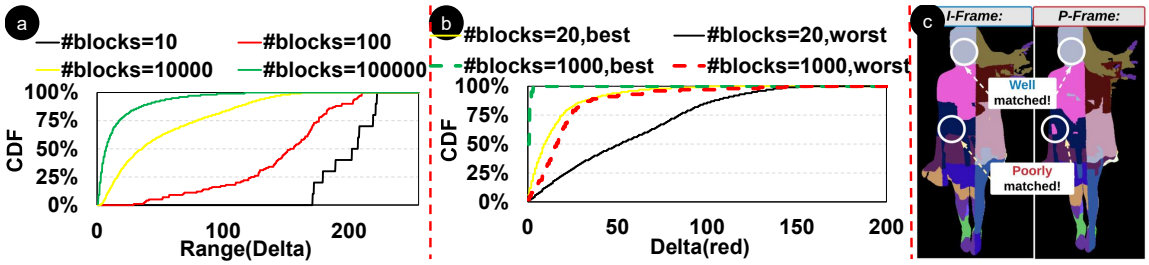


Figure 5.3: a) Spatial locality within one frame. b) Temporal locality among two frames. c) An example of macro blocks segmented using Morton codes in two frames.

application [165] which utilize the Morton code for parallel octree construction[4]; however, we believe ours is the first work that tries to apply such technique in the PCC pipeline.

Morton Code Can Also Assist Attribute Compression: As discussed above, Morton code naturally describes the geometrical relations among points; thus, intuitively, it makes sense to utilize the Morton code to improve the geometry compression. However, our goal is to go beyond just optimizing the geometry compression. Specifically, in the traditional 2D video compression domain, the video frames are usually rich in spatial locality (similar neighboring pixels) within one frame, as well as temporal locality (similar pixel values in corresponding locations across consecutive frames) [166]. This observation motivates us to ask the question: *Do such similarities also exist in the PC streams? If so, can we leverage Morton code (containing location information) to capture such similarities as well?*

Spatial Locality in Attributes: Towards exploring the attribute similarity within one frame, we partition a frame (whose points are first sorted in Morton-code order) from the 8iVFB dataset [8] into 10, 10^2, 10^4 and 10^5 *macro blocks*, plot the CDF of the range for attribute delta ($Max_{red} - Min_{red}$) within one segment/macro block in Fig. 5.3ⓐ, and observe that:

- Overall, with more segments/macro blocks whose size is smaller (compared to a frame), more similarity exists in a block (delta is small). Specifically, compared to the black line (only 10 blocks), the attribute in yellow line (10^4 blocks, each of them is 1000× smaller) exhibits a better similarity (i.e., left-shift towards the y-axis).
- When partitioning the macro blocks in an even more fine grain fashion, as shown in the green line with 10^5 segments, now the CDF curve is pushed towards left even further. This again indicates that, within a smaller macro block, the voxels have richer similarity with their neighbors.

[4]We do not claim the parallel octree construction as our contribution.

Temporal Locality in Attributes: To study the temporal attribute locality across two frames, we plot the CDF of the attribute deltas among two segments in an I-frame and a P-frame in Fig. 5.3ⓑ, and a visual view of how these segments look like in Fig. 5.3ⓒ, and we observe that:

- Compared to the two solid lines (the frame is partitioned into 20 blocks), the two dotted lines (when partitioned into 1000 blocks) are closer to the y-axis, indicating that a finer segment can better capture the temporal-locality.
- Considering the dotted lines with 1000 segments partitioned from I- and P- Frames, the green line represents the smallest delta between two segments, which indicates the upper-bound/the scope of the attribute similarity, whereas the red line represents the largest delta/the least similarity among the segments. Further, the gap between the dotted red and green (1000 blocks) is smaller compared to that between the solid yellow and black lines (corresponding to a 20 block partitioning), thus indicating that a finer partition granularity can observe less variance in the temporal locality opportunities (i.e., the smaller area, the better).
- Moreover, a vertical line can be drawn in this gap range, i.e., $x = \alpha$, and the macro blocks on the left side have "enough" temporal similarities and thus can be compressed with the I-frame (e.g., simply discard the deltas and represent/compress these blocks by the pointers to the matched I-blocks), whereas those on the right have to employ an extra intra-compression step to further compress the deltas. Note that α can be adjusted based on the application preference, e.g., shifting the $x = \alpha$ line to the right results in more macro blocks in the I-frame being directly reused for compressing the P-frame, i.e., higher compression ratio, with a cost of quality drop (more details in Sec. 5.5).
- Fig. 5.3ⓒ illustrates an example with 20 segments in I-Frame and P-Frame. Due to the limited number of segments, one can observe that some highlighted blocks are not well matched. This again confirms that a finer segment can yield a better temporal locality, as also discussed above.

Takeaway: The Morton codes generated as an intermediate result during geometry compression not only improve the geometry compression by increasing pipeline parallelism, but also help to capture/identify the attribute similarities within a frame as well as across frames. Motivated by this observation, we next propose schemes that can utilize Morton codes for *both* geometry and attribute compression in point clouds.

5.4 Intra-Frame Compression Design

As discussed in Sec. 5.3.1, PCC takes several seconds to execute, which is significantly higher than the ideal/real-time demand ($100ms$). The main reason for this inefficiency is the "sequential updates" in the state-of-the-art octree-based algorithms [98] (illustrated in Fig. 5.2). Furthermore, we also observed in Sec. 5.3.2 that Morton codes can reveal opportunities for both geometry similarity (owing to the fact that the Morton code itself is the reflection of the geometrical relationship between points) and attribute similarity (the RGB attributes of two adjacent points are more likely to be similar). Unlike the prior octree-based works [110, 164] which mainly focus on the compression efficiency (i.e., attaining higher compression ratio and good quality simultaneously) with sequential updates and longer execution latency, in this work, we focus primarily on speeding up the PCC at the *edge* and achieving the real-time target mentioned above without losing much quality or compression ratio.

5.4.1 Intra-frame Compression

In this subsection, we first present the state-of-the-art intra-frame geometry and attribute compression techniques and discuss their inefficiencies. We then introduce our proposed intra-frame geometry and attribute compression schemes which are discussed in detail in Sec. 5.4.2 and Sec. 5.4.3.

5.4.1.1 Prior Intra-Frame Compression Inefficiencies

State-of-the-Art Intra-Geometry Compression[5]: As discussed in Fig. 5.2 and Sec. 6.3, most of the existing G-PCC techniques [110, 164] are based on octree data structure. We illustrate the generic pipelines (for ① geometry compression, and for ② attribute compression) employed by the state-of-the-art intra-frame compression techniques in Fig. 5.4ⓐ and 5.4ⓑ. Specifically, the SOTA geometry compression pipeline includes five stages which can be summarized as follows:

- *Raw Frame (Input):* The input raw PC frame contains several (usually millions of) points, carrying both geometry and attribute information. Only the geometry data are forwarded to the upper geometry compression pipeline.

[5]We consider the octree-based technique [110, 164] and RAHT [99, 110] as SOTAs for geometry and attribute compression respectively.

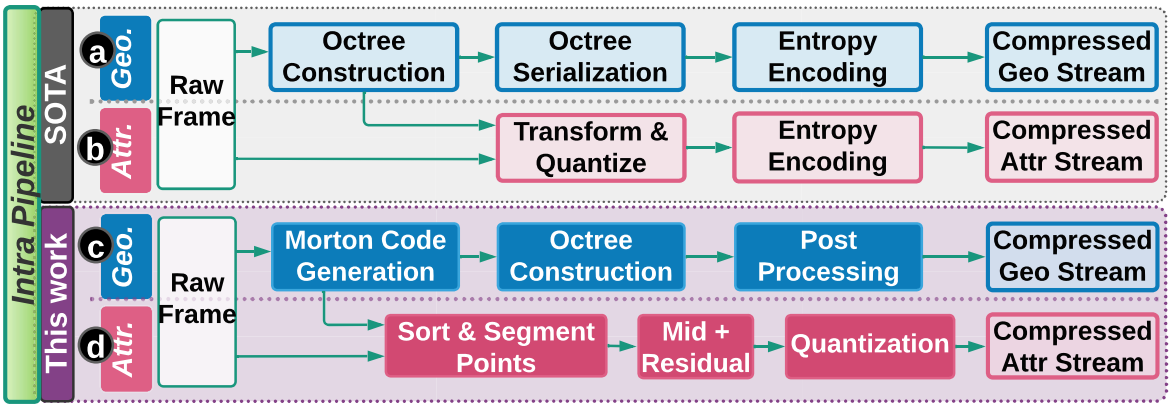


Figure 5.4: Intra-frame PCC pipelines.

- *Octree Construction:* With the input geometry data, the octree construction algorithm is invoked to add the points and update the tree (e.g., the maximum depth required for inclusion of a point, occupancy information for nodes, etc.) in a point-by-point fashion. This point-by-point "update" makes this stage difficult to parallelize.
- *Octree Serialization:* After the octree has been constructed, the tree is traversed in a top-to-bottom manner, in order to extract the *occupy* bits for each node, and record them in a predefined order (e.g., via depth-first traversal), such that the decoder can recover the octree with these occupy bits as well as the serialization order. Note that this step is also time-consuming, as shown in Fig. 5.2. This is because that all the nodes in the tree are traversed sequentially.
- *Entropy Encoding:* To further compress the generated occupy bits vector, a typical encoding technique – Entropy Encoding [167, 168] – is employed.
- *Compressed Geometry Stream (Output):* The final compressed geometry output stream is ready to be stored in the memory or streamed over the network.

State-of-the-Art Intra-Frame Attribute Compression: As shown in Fig. 5.4b, to compress the attribute data, similar steps – *Raw Frame Input*, *Attribute Transform and Quantize*, *Entropy Encoding*, and *Compressed Attribute Output Stream* – are employed. Only the ***Transform and Quantize*** step differs from the geometry compression pipeline, which takes both the raw frame's attribute data as well as the constructed octree as its inputs. With these inputs, the *Transform* step performs linear transformations on the attribute data of each voxel pair (the voxel in level_n and its siblings along x, y, and z dimensions) to obtain a low-pass component and a high-pass component. The high-pass component is quantized and entropy encoded, while the low-pass component proceeds to the next level (level_{n-1}) and serves as a prediction for the voxel's attribute in this upper

level [99]. Note that, this step also needs to be performed sequentially across the tree layers.

Takeaway: The prior octree-based works for both geometry and attribute compression suffer from performance inefficiencies, mainly because the octree construction, serialization and attribute transformations involve sequential computations. To improve the performance, next we want to explore the hidden spatio-temporal locality opportunities (missed by the prior works), and speed up *both* the geometry and attribute compression from *both* the intra- and inter-frame perspectives.

5.4.1.2 Optimizing the Intra-Frame Compression

Driven by the observations above, next we relax the "sequential update" approach that exists in the prior works, and employ the (intermediate) generated *Morton Codes* to reveal the "hidden parallelism" opportunities for compressing a PC frame (shown in Figs. 5.4ⓒ and 5.4ⓓ for the geometry and attribute compression, respectively).

Proposed Intra-Frame Geometry Compression: As can be seen from Fig. 5.4ⓒ, the modified components in our pipeline compared to the previously-proposed geometry compression approach (depicted in Fig. 5.4ⓐ) include the following:

- ***Morton Code Generation:*** Given the raw PC, instead of constructing the octree point-by-point, now the first step is to generate the Morton codes in one shot (note that this can be performed in parallel and only takes $0.5ms$). This additional pre-processing step can draw an overall layout for all the points, which will further help to parallelize the octree construction.
- ***Octree Construction***[4]***:*** Using the Morton codes generated in the previous step, now the octree can be constructed in parallel by employing techniques similar to [113, 169]. Note that this step is slightly different from the one in the prior pipeline shown in Fig. 5.4ⓐ. Instead of updating and storing the occupy bits for each node during the process of adding points, now the outputs of this step are several arrays (Morton codes array, parent array, etc.), which reveal the geometrical relationship across the nodes.
- ***Post Processing:*** Using these relationship arrays, the final step is to post-process them to obtain the occupy bits for each node, and output the compressed geometry stream.

Proposed Intra-Frame Attribute Compression: As shown in Fig. 5.3 and discussed in Sec. 5.3.2, apart from identifying the locality existing in geometry data, the Morton codes can also help us capture the attribute locality. Motivated by this, we further

optimize the attribute compression pipeline for a given frame, as shown in Fig. 5.4ⓓ. Specifically, our proposed new pipeline includes the following three steps:

- ***Sort and Segment Points:*** Unlike the prior works which utilize the octree to capture the spatial locality between the points when compressing the attributes, we use the Morton codes to cluster the points which are spatially close to each other. Specifically, with the Morton codes for all the points (i.e., the intermediate results from geometry compression without any additional overhead), we first sort these points in the Morton code order, and then segment these sorted points into several blocks which can help to gather the points with similar positions/coordinates into one segment.
- ***Mid + Residual:*** Within each segment, since the points are located in small regions, their attribute values tend to have similar numbers. Therefore, instead of recording the exact attribute values for all the points within a segment, we only need to find the "*median value*" of these attributes (as base) and then compute and compress the *residual values* (as deltas) for these points. Fortunately, these computations are light-weight, and can be performed in parallel.
- ***Quantization:*** Finally, these small residual values are quantized to further improve the compression ratio.

5.4.2 Intra-Frame Geometry Compression

In above section, we have discussed our overall proposals for both geometry and attribute compression pipelines. And, as will be shown later in Sec. 5.6, such optimizations are able to bring around 37× speedup w.r.t. the state-of-the-art techniques (1.55s latency in prior works vs. 42ms latency in ours). To better understand where the benefit comes from, next we go over a simple example, given in Fig. 5.5, and answer the following three critical questions: *i) how to increase parallelism for the bottleneck steps?*, *ii) how to integrate such optimizations into the entire geometry compression pipeline?*, and *iii) what are the resulting benefits and overheads?*

5.4.2.1 How to Increase Parallelism?

Fig. 5.5 shows an example of geometry compression. Specifically, there are three points in the this frame: P_0's coordinates are $[0, 0, 0]$, P_1's are $[-1, 0, 0]$, and P_2's are $[3, 3, 3]$. Consider the geometry compression pipeline in PCL [164], where the points are added one-by-one when constructing the octree. Initially, the bounding box is infinitely small (side length is 0, corresponding to no data); and the octree only has one *root* node, which

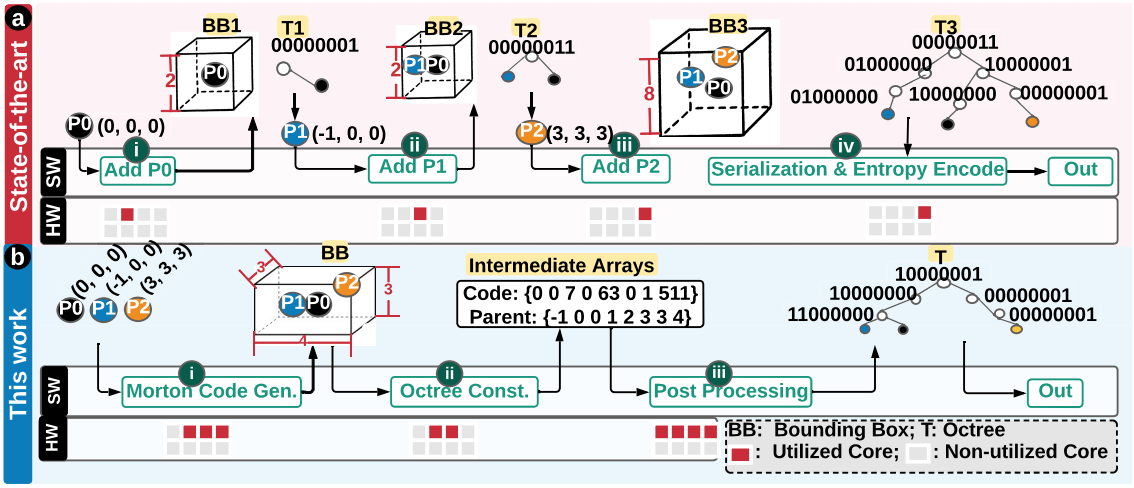


Figure 5.5: Intra-Frame geometry compression example.

is just a "virtual placeholder" containing no information. When inserting the first P_0 point into the bounding box and the octree, two actions are taken: (1) expanding the bounding box with a step-size of 2^n, where $n = 1, 2, 3, \cdots$, until P_0 is wrapped inside the bounding box. In this case, the side length of the bounding box cube becomes 2, and now P_0 is located inside the bounding box. And (2), inserting the new point into the current octree. In this case, P_0 is located in the $7th$ child of the *root* node, and the *root* node now stores the occupy information, which is 00000001 (the right-most 1 indicates a "child" in $7th$ leaf node). Similarly, P_1 is located inside the current bounding box and inserted into the octree as the $6th$ child of the *root*. Interestingly, in order to include P_2, the current bounding box has to expand its side length by 4×, i.e., enlarging from 2 to 8, and now the octree also contains more levels with all three points being in its leaf level. Obviously, both the bounding box and the octree are updated point-by-point, which forces the pipeline to be sequential.

In our proposal shown in the lower figure, instead of constructing the octree in a point-by-point fashion, we process all three points as one "batch" in the *Morton Code Generation* step in parallel, and output the final bounding box cuboid with side lengths 4×3×3 (x-axis: 3-(-1) =4, y-axis: 3-0 =3, and z-axis 3-0 =3). With the Morton code in place, next we invoke the parallel octree construction technique (for more details, please refer to [113, 169]) to construct the octree. Note that this step is also amenable to parallelism.

5.4.2.2 How to Apply to PCC?

As discussed in Sec. 5.4.1.2, the *octree Construction* step returns several arrays containing the relationship among octree nodes; e.g., the *code array* contains the Morton codes for all the nodes, while the *parent array* contains the index of the current node's parent in the code array (e.g., in Fig. 5.5, -1 in the parent array means that the root node has no parent, whereas $parent[7] = 4$ means that for the $7th$ node (whose code is $code[7] = 511$), the index for its parent node in the code array is 4 (whose code is $code[4] = 63$)). Although such arrays already contain the necessary information to decode the geometry information for the PC, they are *not suitable for compression tasks*. For example, to store/transmit these two arrays, $4bytes \times 16 = 64bytes$ are needed. However, even without any compression, we only need $4bytes \times 3 \times 3 = 36bytes$ to represent these 3 points. Therefore, an extra post-processing step is needed to merge these two arrays in the "occupy bits" style. As shown in Algo. 4, to obtain the occupy bits for one branch node, we first calculate which branches its children should be on (e.g., $C[j]\%8$ in Line #5), and then merge all of its occupied branches via the "|" operation. This inexpensive post-processing step can be applied to all branch nodes in parallel, thus does not bring much overhead.

Algorithm 4: Octree Occupy Bits Generation Algo.

Input : C: Code Array; P: Parent Array; N: Number of Points

1 Occupy Bits Array: $O = \{\}$
2 $L = len(C) - N$
3 **for** i *in* L **do**
4 $\quad p = P[i]$
5 $\quad O[p] \mid = (C[j]\%8),\ P[j] = p$

Output : O: Occupy Bits Array

5.4.2.3 What are the Benefits and Drawbacks?

To summarize, compared to the prior schemes like PCL [164] and TMC13 [110], the most obvious benefit from our proposal is the potential performance improvement in terms of latency and energy savings, due to embracing more parallelism. In fact, the CPU-based PCC pipeline in PCL [164] requires $O(N \times D)$ time complexity to process N points, with a D-layer tree. In comparison, given a GPU-based system with k parallel cores, our design requires only $O(\sum_{i=1}^{D} N_i/k)$ time (N_i is #nodes in layer$-i$). And, as we show later in Sec. 5.6, our results indicate 37× speedup for the geometry compression. Apart from compression, the de-compression stage can also be run in parallel after

applying our proposal, and can be even faster than the compression stage due to its reduced complexity (note that, this also applies for the attribute compression proposals as discussed later in Sec. 5.4.3.1 and Sec. 5.5). Specifically, with our current "sub-optimal" implementation (e.g., the codes are not fully optimized), the de-compression stage (including both geometry and attribute de-compression) for Redandblack video [3] only takes $\approx 70ms$ per PC frame, which is less then the PC compression latency as we will discuss later in Sec. 6.6.2.

Such significant speedup comes with a reduction in quality. In the example shown in Fig. 5.5, the octree constructed based on the Morton codes is slightly different from the one generated by the sequential algorithm (which is lossless). In fact, in our octree, the P_0 node now contains geometry information of $[-0.43, 0, 0]$ (-0.43=-1+1/7×4), which is slightly different from the original $[0, 0, 0]$, whereas the other two points, P_1 and P_2, are exactly same as the original ones. Thus, as we discuss later in Sec. 5.6, our proposal drops the quality a little bit (PSNR $\approx$ 80dB in our design). This quality degradation comes as a result of the parallel algorithm, and we argue that the PSNR values resulting from our proposal are still very good for most video and AR/VR applications [170, 171].

Another comparison parameter is the *compression ratio*. Our proposal provides similar compressed size as TMC13 [110] ($\approx 0.1\times$ larger) when exploiting the entropy encoding. However, this entropy encoding consumes $\approx 100ms$, which halves our performance gains. Thus, in order to harvest most of the speedup benefits ($42ms$ vs $1.55s$), in our design, we discard the entropy encoding and still achieve reasonable compressed size, which is $\approx 0.5\times$ larger than that of TMC13 [110].

5.4.3 Intra-Frame Attribute Compression

The above discussion has explored the opportunity of utilizing Morton codes to speedup the geometry compression. Recall from Fig. 5.3ⓐ that, there also exists spatial locality for attribute compression, which can be identified with the help of the Morton code. Let us now consider the example in Fig. 5.6 with three points – P_0 with geometry data of $[0, 0, 0]$ and one attribute value of 50 (in this example, we set the attribute as a scalar for simplicity; normally, the attribute should be a vector, e.g., RGBs), P_1 with a $[-1, 0, 0]$ geometry and 52 as the attribute, and P_2 with a $[3, 3, 3]$ geometry and 54 as the attribute. Here, we assume that the octree for these points has already been established using the geometry pipeline, and we next go through our proposed attribute pipeline step by step and explain where the envisioned benefits will come from.

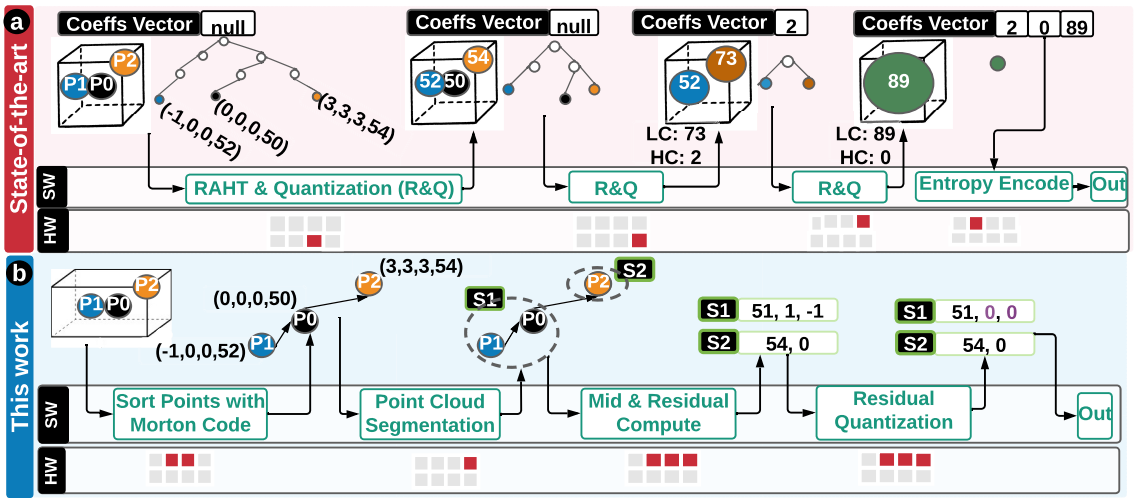


Figure 5.6: Intra-Frame attribute compression example.

5.4.3.1 How to Speedup?

First, RAHT [99] takes the initial octree (which is deepest now) as input, and invokes *RAHT and Quantization* to perform the linear transformation on the leaves with their siblings along the x, y and z dimensions, and shrinks the tree layer-by-layer. Each transformation emits out one low-coefficient (LC) and one high-coefficient (HC) by the following equation:

$$\begin{bmatrix} LC \\ HC \end{bmatrix} = \frac{1}{\sqrt{w_1 + w_2}} \begin{bmatrix} \sqrt{w_1} & \sqrt{w_2} \\ -\sqrt{w_2} & \sqrt{w_1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2, \end{bmatrix} \tag{5.1}$$

where w_1 and w_2 are the weights for the two leaves (#occupied voxels in this leaf), and a_1 and a_2 are the attributes. Now, the HC is quantized and entropy encoded, while the LC is further sent to the next *RAHT and Quantization* round and serves as the attribute of the large voxel/leaf in the upper layer. This procedure is repeated until we reach the root node. In this example, eventually the coeffs vector contains $[2, 0, 89]$, which can be further compressed by entropy encoding. This entire pipeline also requires sequential processing across the octree layers, which is obviously time-consuming when the number of points is large and the octree is deep. In fact, our profiling shows that RAHT takes around 2 seconds to process a typical frame with around 1M points, on a typical edge device.

Towards addressing this significant performance inefficiency, we investigate the insight from the above discussion in Sec. 5.3.2, which indicates that the locality revealed by the Morton codes does not only exists among the geometry data, but it can also help with

the attribute compression. Note that, the Morton codes have already been generated as intermediate results during the geometry compression, thus, can be (re)used to identify the spatio-locality for the attribute compression without any extra cost. Specifically, as shown in Fig. 5.6, in order to capture the spatial locality in attributes (e.g., points with similar Morton codes tend to have similar colors), our proposed pipeline first sorts the points using the Morton codes and then partition/group them into multiple segments. The points within one segment are geometrically close to each other, and hence their attributes are also likely to be similar. Thanks to this, for each segment, we just need to store one *medium* value as *Base* and several *residual* values as *Deltas* (which are mostly small, due to similarity). And finally, we quantize these deltas for achieving higher compression ratio. In this example, two vectors (as there are two segments) store the final data, including $Mid = 51, Delta = [0, 0]$ for the first one segment, and $Mid = 54, Delta = [0]$ for the second.

5.4.3.2 What are the Pros and Cons?

As indicated in Fig. 5.6, compared to RAHT, our proposal reuses the intermediate Morton codes, which have been computed during the geometry compression, to precisely identify the points with similar attributes from a set of irregular points. This is expected to be much faster than RAHT, and in fact, our experimental results show $\approx 49\times$ speedup ($53ms$ vs $2.6s$).

However, as also shown in Fig. 5.6, the storage size after our compression is larger than RAHT, since each segment requires one vector storage to store its median/base and (quantized) delta values. Although the $48\times$ speedup brought by our proposal is promising in terms of performance gain, the observed $2\times$ compression inefficiency needs to be addressed.

5.4.3.3 How to Further Improve the Compression Efficiency for Attributes?

Towards further improving the compression efficiency, one could consider different options. Instead of throwing more compute power, we want to emphasize that, the discussion in this section only focuses on the attribute locality within one frame, which has ignored the potential localities among consecutive frames. In fact, if frame-2 does not vary much with respect to frame-1, intuitively, there would be temporal locality between the two frames. Thus, to further improve the attribute compression efficiency, in the next section, we investigate the inter-frame similarity opportunity.

5.5 Inter-Frame Attribute Compression Design

Motivated by the above discussion, in an attempt to further improve the compression efficiency from an inter-frame perspective, in this section we explore the "attribute similarity" that exists across consequent frames in a PC video, and explain the design details of our proposed inter-frame attribute compression scheme. Similar to the intra-frame proposals discussed in Sec. 5.4, we again use a simple example using a state-of-the-art inter-frame compression technique [114] and our proposal, and study the following important questions: i) *what* is the opportunity?, ii) how do we *capture* and *exploit* such opportunity?, and iii) what are the potential *benefits*?

5.5.1 Inter-Frame Attribute Compression

5.5.1.1 What is the Temporal Opportunity?

As we have shown earlier in Fig. 5.3❻, two blocks (a set of points) which are located close to one another are likely to contain similar color pixels. In the example shown in Fig. 5.7, the first frame, I-Frame, contains three points – P_0 with geometry data $[0, 0, 0]$ and an attribute value 50, P_1 with $[12, 8, 13]$ for geometry and 52 for attribute, and P_2 with $[19, 26, 58]$ for geometry and 20 for attribute. Obviously, the two P_0 points in I-Frame and P-Frame are exactly the same, which could be completely reused during the compression of the P-Frame. Moreover, the two P_1 points are located closely (i.e., $[12, 8, 13]$ vs $[12, 8, 12]$), and contain very similar attribute values (52 vs 51). Thus, the P-Frame can also be further compressed by reusing the P_1 data in the previous I-Frame, without losing too much quality. On the other hand, the two P_2 points are relatively far away from each other and their attribute inputs are quite different, offering little reuse opportunity. To summarize, in this example, the first two points in I-Frame, P_0 and P_1, could be reused for compressing the P-Frame, thus reducing the compressed output size.

5.5.1.2 How to Capture the Temporal Opportunity?

To identify such similarity/reuse opportunities across frames (shown in Fig. 5.7), the macro-block based state-of-the-art approach [102] first needs to generate two macro block trees (where the minimum voxel dimension in this tree is of a predefined size) – one for I-Frame and the other for P-Frame. Next, for each leaf node/block in the P-MB-Tree, the entire I-MB-Tree needs to be traversed in a top-to-down fashion, and the exactly-matched leaf in the I-MB-Tree is found. In this case, the found leaf $L1$-I, which contains two

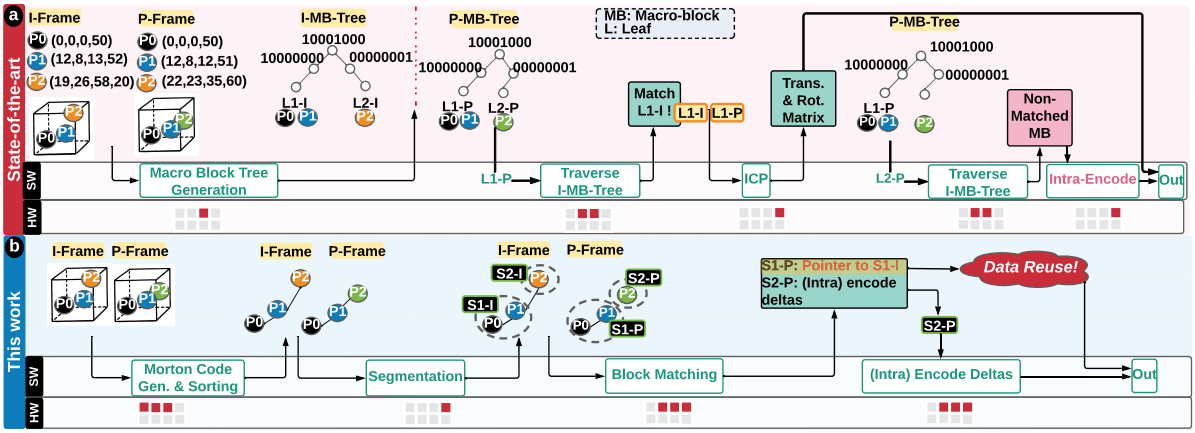


Figure 5.7: Inter-Frame attribute compression example.

points (P_0 and P_1), is a perfect match; however, no match can be found for the $L2$-P leaf. This process is repeated for $O(N)$ times, where N is the number of macro-blocks in the P-Frame. This processing can be quite time-consuming, and our profiling shows that it usually takes $\approx 5.9s$ to compress one predicted PC frame even when running on 4 CPU threads.

Instead, our proposal takes advantage of the Morton code generated in the geometry compression, which is a good indicator for attribute similarity (as discussed earlier in Sec. 5.3.2). Specifically, our proposal consists of the following 4 steps:

PC sorting: As shown in Fig. 5.7, compared to the irregular raw PC, after sorting the PC via Morton-code, the adjacent points are more "regular" and geometrically closer, and thus share rich attribute similarities.

Segmentation: The next step is to partition the sorted PCs (i.e., I-frame and P-frame) into several blocks/segments (similar to the term "macro-blocks" in 2D image encoding).

Block/segment match (BM): for each block in the P-frame, we iterate through all the candidate blocks in the I-frame and calculate the difference between these $< I, P >$ block pairs. Finally, the candidate I-block which differs minimally with the P-block is picked as its "best-matched/reference" block. Specifically, given two blocks with K-points, we use 2-norm attribute distances (see Equ. 5.2) to measure their difference:

$$Diff(I_{block}, P_{block}) = \sum_{i=1}^{K}(r_{iP} - r_{iI})^2 + (g_{iP} - g_{iI})^2 + (b_{iP} - b_{iI})^2 \quad (5.2)$$

where $P_{block} = \{(x_{iP}, y_{iP}, z_{iP}, r_{iP}, g_{iP}, b_{iP})\}$, $I_{block} = \{(x_{iI}, y_{iI}, z_{iI}, r_{iI}, g_{iI}, b_{iI})\}$, for $i = \{1, ..., K\}$. Note that, the BM can be performed in parallel as

there is no dependence across blocks.

Reuse: for blocks for which the reference blocks are similar enough (e.g., the 2-norm differences are less than the pre-defined thresholds), only the pointers to their reference blocks will be recorded (in our proposal, for each P-block, we set the number of candidate blocks as 100, thus, 6 bits are sufficient for encoding one P-block). On the other hand, if the "best matched I-block" is not as similar as the P-block (e.g., the 2-norm differences are larger than the threshold), simply approximating the P-block with its reference block will significantly degrade the quality; instead, we compute and store the deltas for such block pairs, and then invoke the *Base+Deltas* technique, as mentioned in Sec. 5.4.1.2 for intra-frame compression, to further compress these deltas.

5.5.1.3 What are the Pros and Cons?

The proposed inter-frame attribute compression further improves the compression efficiency by skipping the redundant storage for the same/similar segments matched across frames, with extra latency overhead (but still much better than the state-of-the-art – $139ms$ vs $5.9s$). In this example, for compressing the P-Frame, one pointer (for $S1$ which contains P_0 and P_1) and only one post-intra-encoded compressed delta (for P_2) are required for storage, instead of storing all three. However, the proposed inter-frame compression pipeline has additional steps (PC sorting and block matching), which collectively take about $139ms$ for a typical PC frame.

5.5.2 Combining Inter-frame and Intra-frame Compression

Simply put, our intra-frame compression proposal can significantly reduce the execution latency, while the inter-frame compression proposal can further improve the compression efficiency. We emphasize here that, these two proposals can work in an interleaved fashion (with a frame-level granularity) for a PC video stream. Specifically, in our design, the PC frames are encoded in an "IPP" fashion, where each I-frame is followed by two P-frames. Further, for the P-frame, as discussed above, the pre-defined threshold for determining the block matching (i.e., is the $< I, P >$ block pair good/similar match or bad/dissimilar match) can be tuned according to the application preference, e.g., for applications which favor good quality, more post-intra-encoded blocks (e.g., store and encode the deltas instead of simply reusing) are preferred. On the other hand, for applications that demand very small compressed data sizes to transfer through the network, the threshold condition can be relaxed to favor the *direct reuse* more. For example, in this paper, we pick

two different thresholds to strike a balance between latency, quality and compression efficiency, as will be discussed in detail later in Sec. 6.6.2. We also investigate how different thresholds/preferences shape the behavior of our approach in a sensitivity study in Sec. 6.6.3.

5.6 Experimental Results

In this section, we compare our proposed intra-frame and inter-frame designs against two different PCC techniques, by evaluating four metrics critical for the PC-based applications – execution latency, energy consumption, video quality, and compression ratio. Towards this, we first describe the configurations (Sec. 6.6.1) used for our analyses, e.g., experimental platform, dataset, and different designs (Sec. 5.6.2). We then compare those design schemes on our platform (Sec. 6.6.2). Finally, we provide detailed insights on how to tailor the PCC pipeline to cater to various application preferences (Sec. 6.6.3).

5.6.1 Methodology

5.6.1.1 Evaluation Platform

To evaluate and compare the proposed intra- and inter-compression designs with the state-of-the-art works, we use the NVIDIA Jetson AGX Xavier board [17], which is an edge development board, and is well-known to simulate the realistic edge development environment. Specifically, it is equipped with a 512-core Volta GPU, a 8-core ARMv8 64-bit CPU, and 32GB 256-Bit LPDDR4x Memory. In our implementation, we start the application from CPU (reading the PC data), and then offload the computations to GPU (octree construction, block matching, etc.), and the compute mode of Jetson AGX Xavier board is set to be 15W.

5.6.1.2 Point Cloud Dataset

We use two dynamic PC video datasets – the 8i Voxelized Full Bodies (8iVFB) [8] and the Microsoft Voxelized Upper Bodies (MVUB) [9] datasets in our evaluations. Specifically, we pick four videos from 8iVFB, and two videos from MVUB. The 8iVFB dataset contains the PC data of four persons, captured by 42 RGB cameras placed at different angles, while the MVUB dataset consists of five subjects captured by four frontal RGBD cameras. All

these videos used are captured at 30fps, and voxelized into 1024×1024×1024 voxels (3D points), with each point containing three *float-pointing* coordinates and three *unsigned char* RGBs.

Table 5.1: Six videos in 8iVFB [8] and MVUB [9] datasets used in this paper.

Video	**Redandblack**	**Longdress**	**Loot**	**Soldier**	**Andrew10**	**Phil10**
#Frames	300	300	300	300	318	245
#Points/Frame	727070	834315	793821	1075299	1298699	1486648

5.6.2 PCC Design Configurations

To demonstrate the effectiveness of our proposal, we evaluate the following five PCC designs:

- **TMC13** [110]: We use TMC13 (G-PCC codec from MPEG), as the *state-of-the-art* approach for *intra-frame compression*. Especially, by tuning the parameters (e.g., octree depth, compression algorithm, etc.), we utilize the octree-based method to compress the geometry data losslessly, while the attributes are compressed by the predictive RAHT lossily. We use this tool to compress every single PC frame, and measure the encoding latency, energy consumption, the compressed stream size, and finally measure the quality (PSNR) of the decoded frame by *pc_error_d* tool [172].
- **CWIPC** [102,114]: CWIPC is a PCC library that supports the inter-frame compression (encoding the predicted frame via macro block (MB)-based motion estimation). We use CWIPC as the *state-of-the-art* approach for *inter-frame compression* and build it with multi-thread option. In our setup, one I-frame(intra-compressed frame) is followed by two P-frames(predicted frames), and the number of threads for MB matching is set to 4. Similarly, we have opted to use octree-based algorithm for geometry compression, and directly applied entropy encoding to the raw attributes.[6]
- **Intra-Only**: We apply our intra-frame compression method discussed in Sec.5.4 to each of the PC frames. Specifically, we choose to segment each PC frame into 30000 blocks[7], and use a 2-layer encoder (more specifically, we first encode the attributes via the proposed intra-frame encoder as described in Sec. 5.4, and then treat the obtained delta values as new attributes, and finally feed them again to the encoder to further increase the compression efficiency).

[6]Based on our profiling, the provided attribute compression APIs (e.g., JPEG-Turbo-based) would degrade the quality of PC significantly; thus, we do not use such APIs/Libs in our evaluations.

[7]We decide the parameters for intra- and inter-compression by profiling several frames in the 8iVFB [8] dataset, to obtain a relatively balanced design point between compressed size and quality.

- **Intra-Inter-V1**: As mentioned in Sec. 5.5, only part of the blocks/segments can be directly approximated by the reference block. Therefore, we add one more parameter to control the ratio of such blocks. Especially, for each P-block, we first calculate its 2-Norm distance to its *best matched block* in I-frame. We then compare this distance with a preset threshold (e.g., 300^7) to determine if this segment can be approximated by *direct reuse*; otherwise, we mark it as an post-intra-encoded block. Also, the total number of blocks is 50000, and the search step is set to be the size of the current P-block (i.e., to find the best matched block for current P-block, each time, we traverse the search region in the reference frame by this step size).
- **Intra-Inter-V2**: For this configuration, we choose a larger threshold (1200) for the "direct-reuse decision making" such that the ratio of the *direct reuse* will increase with slight drops in quality. Other settings are not changed (they remain the same as in the Intra-Inter-V1 version).

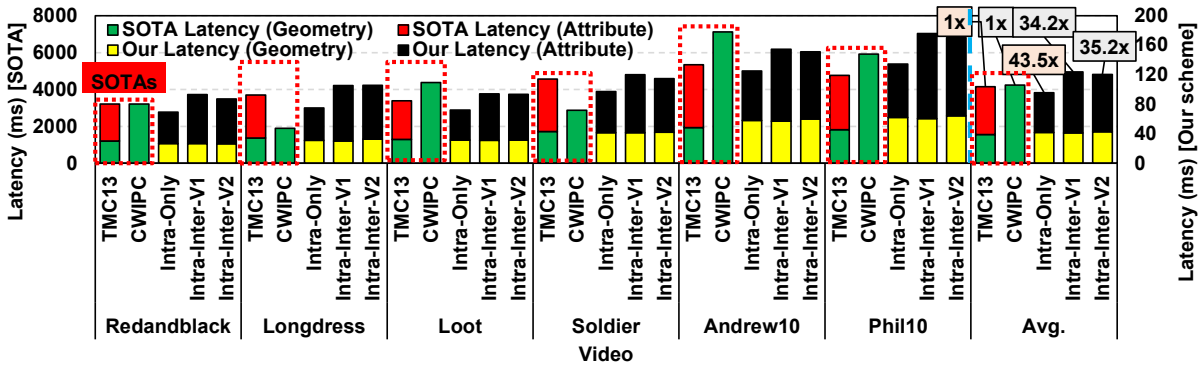
5.6.3 Results

We first compare the execution latency, energy consumption, quality (PSNR [peak signal-to-noise ratio]), and compression efficiency (compressed size) in Fig. 5.8, when using various designs explained in Sec. 5.6.2 to compress the six PC videos from the 8iVFB and MVUB datasets (described in Table 5.1). Then, we discuss and present the validity of our results on the smartphones.

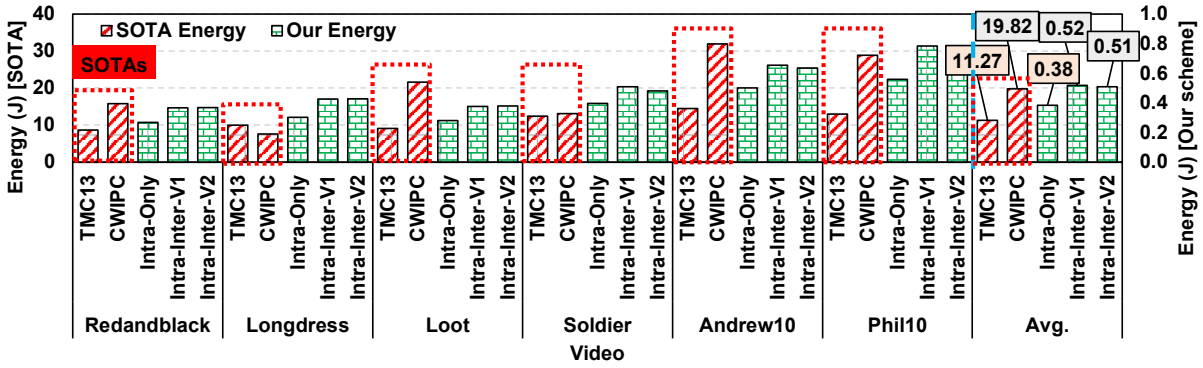
Execution Latency: We first compare two SOTA schemes (TMC13 [110] for intra-frame compression, and CWIPC [114] for inter-frame compression) with our three proposals (intra-only, (better) quality-oriented Intra-Inter-V1, and (better) compression-oriented Intra-Inter-V2), and present the collected execution latencies in Fig. 5.8a. For each video, these five designs are listed on the x-axis. The primary y-axis (left) shows the latency in ms for SOTAs, whereas the secondary y-axis (right) gives the latency in ms for our proposals.

From this plot, the following observations can be made:

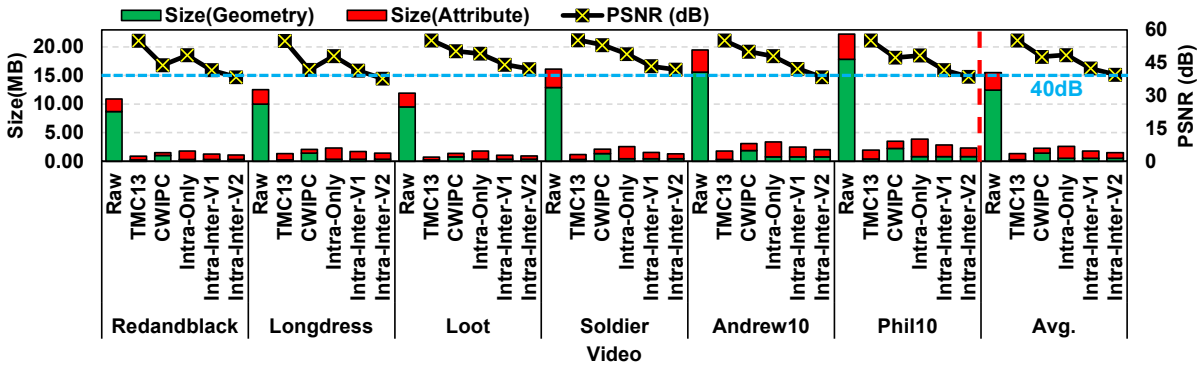
- *TMC13:* TMC13 takes around $4152ms$ to compress one PC frame, including $1552ms$ for geometry compression and $2600ms$ for attribute compression.
- *CWIPC:* CWIPC takes about $4229ms$ (mainly for geometry compression as the attributes are directly entropy-encoded without any other efforts, as mentioned in Sec. 5.6.2).
- *Our Intra-only:* The performance of above two techniques has two orders of gap with



(a) Latency breakdown for geometry and attribute compression



(b) Energy consumption



(c) Compressed size & PSNR

Figure 5.8: Results: (a) Latency breakdown. (b) Energy consumption. (c): Compression efficiency.

the $\approx 100ms$ real-time requirement [97]. On the other hand, our intra-only scheme takes only $95ms$ ($42ms$ for geometry and $53ms$ for attribute compression), which is $43\times$ faster w.r.t. TMC13. This is due to: 1). the octree can be constructed in parallel with the help of Morton codes, which speeds up the geometry compression by $37\times$; 2). by utilizing the spatial locality with Morton code for attribute compression, instead of performing transforms through octree layers, in our proposal, only simple subtractions

are needed for computing deltas. Further, all the points are processed in parallel; 3). we discard the entropy encoding for further speedup.

- *Our Intra-Inter-V1 (Quality-oriented):* This design favors the quality over compression efficiency, and it only takes $124ms$ ($41ms$ for geometry compression, and $83ms$ for attribute compression), contributing to around 34× speedup w.r.t. CWIPC. This speedup comes from: 1). instead of searching the matched macro block from the entire space (the I-MB-Tree traversal as described in Sec. 5.5), given the points sorted in the Morton code order, the search space for block matching in our proposal is minimized (in the I-frame, now, we only need to search the neighboring regions for the current P-block); 2). for the matched macro block, instead of executing the complex iterative closest point (ICP) [173] algorithm for the translation and rotation matrix, we only need to record a pointer to the matched block in I-frame, without any extra computation overhead.
- *Our Intra-Inter-V2 (Compression efficiency-oriented):* Similarly, our Intra-Inter-V2 scheme (oriented towards high compression ratio) takes $121ms$ ($43ms$ for geometry and $78ms$ for attribute compression), which represents about 35× speedup w.r.t. CWIPC. There is not much performance difference between V2 and V1, because we have to run the block matching algorithm on all the blocks before making the "direct-reuse" decision, which dominates the entire pipeline in both the design variants.

Energy Consumption: Fig. 5.8b plots the energy consumptions (in J) for the SOTA works on the primary y-axis (left), and our proposals on the secondary y-axis (right). Clearly, TMC13 and CWIPC are two of the most energy-consuming schemes, which consume $11.3J$ and $19.8J$, respectively, for one PC frame. This is mainly due to their long execution latencies. Further, as mentioned in Sec. 5.6.2, 4 CPU threads are invoked to perform the macro-block matching in CWIPC, this results in even higher CPU power. For example, the average CPU power for TMC13 is $1687mW$ whereas $3622mW$ for CWIPC. On the other hand, our Intra-Only scheme only consumes $0.38J$ per PC frame, which represents 96.6% energy saving w.r.t. TMC13, while our Intra-Inter-V1 and Intra-Inter-V2 only consume $0.52J$ and $0.5J$ energy, respectively, which translate to $\approx 97\%$ energy savings w.r.t. CWIPC. Note that, although our schemes employ the GPU with an extra overhead (e.g., the GPU power is about $1065mW$), the CPU power is reduced (e.g., around $1310mW$, lower than that in TMC13 and CWIPC) since most of the computations are offloaded to GPU. Therefore, the overall energy savings brought by our schemes are similar to the corresponding execution latency reductions discussed above.

Compression Efficiency: To investigate how the compression efficiency changes with the above schemes, in Fig. 5.8c, we plot the compressed size (in megabytes) shown on the primary y-axis (left), and the PSNRs (in dB) for attributes[8] on the secondary y-axis (right), and observe that:

- *TMC13:* compresses the input frame size to be only 8% of the original while preserving the best video quality (PSNR is 55dB). This is mainly because TMC13 performs lossless geometry compression and almost-lossless attribute compression in our settings. Additionally, the transform and quantization in TMC13 can output (near-)zero coefficients, which significantly increases the compression ratio. Note however that, this comes at the cost of longer processing latency, as shown in Fig. 5.8a.
- *CWIPC:* Overall, when employing the CWIPC, the output frame size reduces to around 14% of the original input frame (including 63% of geometry and 37% of attribute data). The reason for such low compression ratio is because – ① for intra-attribute compression, only entropy encoder is applied; and ② even with the inter-frame compression, only few macro blocks are matched and inter-encoded, which limits the benefits from inter-frame compression. As for the quality, it drops the PSNR by 7.2dB when compared to TMC13, due to the macro block-based approximation for the inter-frame compression.
- *Our Intra-only:* this design emits out a compressed frame with $\approx 17\%$ of the original data size (including 19% of geometry and 81% of attribute) and provides PSNR values up to 48.5dB (only 6.5dB drop compared to TMC13). Clearly, the geometry data has been compressed very well, as opposed to the attribute data. Recall from our intra-frame design discussion in Sec. 5.4 that, the Morton codes can precisely describe the geometry relations among points (thus, ensuring a good geometry compression), but sometimes they may not work well for the attributes, especially when the spatial locality is not rich for some blocks/frames.
- *Our Intra-Inter-V1 (Quality-oriented):* To further reduce the frame size, this design exploits the temporal locality across PC frames. As a result, the data size becomes 5% less than our intra-only design (e.g., only 12% of the original size), while dropping the quality by 6.1dB due to the block-level approximations in the inter-frame compression.
- *Our Intra-Inter-V2 (Compression efficiency-oriented):* Similarly, this design further reduces the compressed data size by 2%, by adjusting the threshold for "direct-reuse decision making", as discussed in Sec. 5.6.2. At the same time, the quality further

[8]The geometry PSNR are excellent for all designs (e.g., $>$ 70dB), so we only compare the PSNR for attributes in the evaluations.

drops by 2.9dB. Still, we argue that, even with the Intra-Inter-V2 option (see one demo in Fig. 5.10a(iv)) which has the worst quality, the absolute PSNR is close to 40dB, which is sufficient for many of the video applications that do not require very high resolution [171]. For other high-demanding applications like AR-based surgery [174], our proposals may jeopardize the video quality, and consequently, we may need further software and/or hardware optimizations for improved user experience.

Correlation with the evaluations on smartphones: Based on our profiling, the power consumption of our proposal is $\approx 4W$, which is below the peak discharge power of modern smartphones (10W [175, 176]), meaning that our proposal will work fine on smartphones as well. To further prove this, we also change the compute mode of Jetson AGX Xavier board to 10W, and measure the execution latency for loot video [4]. We observe that the total execution latency when using 10W mode is 1.29x of that when using 15W mode (the mode for collecting the main results). Such similar performance demonstrates that our proposal is expected to work well for low-power edge devices like smartphones as well.

5.6.4 Architectural Insights:

In this section, we further investigate the energy efficiency characteristics of the proposed optimizations by dissecting the total energy consumption for the inter-frame attribute compression proposal (which is the most time- and energy-consuming one among the proposed geometry, intra- and inter-frame attribute compression techniques). We then analyze the bottleneck of our proposal and provide insights for potential architectural support to make PCC even more energy efficient. As shown in Fig. 5.9, the address generation stage for storing P-blocks' deltas/residuals consumes 32% of total energy, while the computation for the 2-norm attribute distance consumes 51% energy, which dominates the total energy consumption, and is, therefore, our target for next-step optimization. Specifically, to compute the 2-norm attribute distance, two kernel functions are invoked (*Diff_Squared* and *Squared_Sum*), which consume 35% and 16%, respectively of the total energy. Such high energy consumption of these two kernels can be attributed to two reasons. First, these are the most frequently invoked kernels during the block matching stage. Second, processing a typical PC with $1M$ points in a fully-parallelized fashion poses very high demands on the GPU resources (e.g., the number of available threads or the memory budget), which are quite limited, especially in edge devices. More interestingly, software-level optimizations for this step have been fully exploited (e.g., the kernel functions are invoked in a fully-parallelized manner), yet it still dominates the

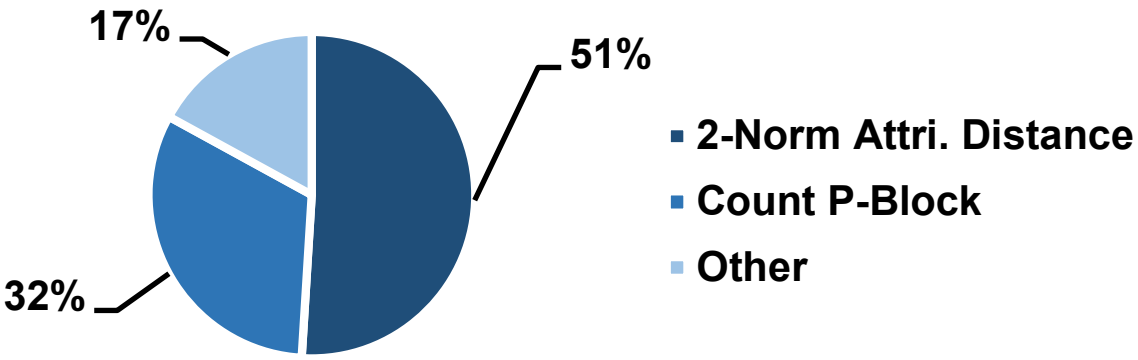
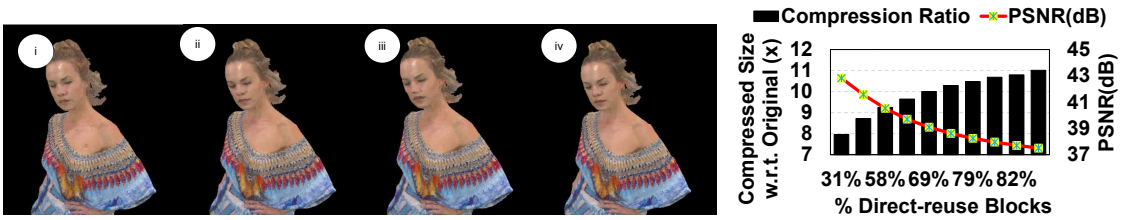


Figure 5.9: Energy consumption breakdown for inter-frame attribute compression for Loot video [4].

latency and energy. This motivates us to further look into architecture-level optimizations in future work, including 1) replacing GPU with ASIC to improve the power efficiency for *Diff_Squared* computation kernel; 2) customizing the accelerator (e.g., number of layers of the tree-structured adder) for the *Squared_Sum* kernel; and 3) minimizing data movements such as inter-SoC (e.g., between GPU and CPU) or intra-SoC (e.g., across L2/L3 caches in a GPU) memory copies.

5.6.5 Sensitivity Study

Our proposed intra-frame PCC utilizes the Morton code to capture the spatial locality, and significantly speeds up the compression (44×), with high compressed quality (48.5dB PSNR). Additionally, by exploiting the temporal locality across frames, our inter-frame compression further increases compression efficiency with the cost of longer processing latency and lower quality. To study how the inter-compressed frames/blocks would affect the compression efficiency (the compressed size w.r.t. the raw PC frame) and the quality (PSNR), we *reconfigured* the number of "direct-reuse" blocks by adjusting the threshold as discussed in Sec. 5.6.2. As shown in Fig. 6.15, with fewer "direct-reuse" blocks (e.g., only 31% of the I-blocks are directly reused in the left-most bar), the PSNR drops slightly when compared to the intra-frame compression, while the compression ratio is also the worst. On the other side, by increasing the percentage of the "direct-reuse" blocks, the compression efficiency also increases, at the cost of a PSNR degradation (e.g., the PSNR reduces to 38dB with 83% "direct-reuse" blocks). Hence, to enhance the flexibility of our proposed design for trading off the compression efficiency with the quality, we can use the *percentage of "direct-reuse" blocks* as a *tunable design knob*, for which, users can



(a) Original vs. decoded PCs with our proposals.

(b) Sensitivity study.

Figure 5.10: (a) comparison between i: raw PC and our proposals (ii: intra, iii: intra-inter-V1, iv: intra-inter-V2). (b) PSNR v.s. compression ratio (i.e., input size / compressed size).

choose the appropriate value based on their preferences (i.e., fewer "direct-reuse" blocks with higher PSNR vs. more "direct-reuse" blocks with higher compression efficiency).

5.7 Concluding Remarks

PC processing has become the trend for many video applications spanning scientific computing, education, healthcare and entertainment, and is recently being offloaded to the edge. PC compression is an essential component of PC processing (and a critical performance bottleneck), which affects video quality, user experience, and energy efficiency. Unfortunately, prior works mainly focused on compression ratio, but did not consider the performance and energy implications, particularly for edge devices. This paper exploits the data similarity opportunities in *both* geometry and attribute data from *both* intra-frame and inter-frame perspectives, and proposes two complementary designs for minimizing the compression latency and energy requirements for pushing the PC compression to the edge. And, more importantly, these proposals are compliant with the emerging MPEG PCC standards [159]. The experimental results with six PC videos show that our proposals provide 34× speedup (latency reduces from $4.2s$ to $121ms$) and 96% improvement in energy efficiency, with only 13% compression ratio drop and a minimal degradation in video quality with respect to the state-of-the-art schemes. Note however that, even with our proposals, the execution latency per PC frame is still slightly beyond the real-time requirement (i.e., $\geq 100ms$). Towards this, in the future, we plan to explore GPU-specific optimizations such as compile-time instruction fusion for better parallelism, or provide additional architectural support for our proposal by investigating the hardware designs with respect to FPGA modules or customized ASICs, to optimize the bottleneck stage and make PCC on edge devices even faster/more efficient (e.g., $\approx 33ms$ for $30fps$

display refresh rate).

Chapter 6
EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices

6.1 Introduction

Point Cloud (PC), a representation of objects in 3D space using a huge collection of points, primarily used for terrain scanning by military and space agencies and various industrial applications (e.g., engineering, architecture, archaeology, etc.) has gained tremendous attraction over the past few years. Especially, the recent advent of various emerging applications requiring hyper-realistic visualizations of objects such as 360-degree or volumetric video streaming [128, 177] in Virtual Reality (VR), virtual object and real-world interaction in Augmented Reality (AR) [178], cultural heritage modeling and rendering [179], as well as applications demanding high-fidelity version of the physical world such as object detection in autonomous driving and robotics, have put PC in the forefront. Additionally, research advancements in point acquisition technologies (such as 3D scanning [180], photogrammetry [181], etc.) as well as the availability of affordable and convenient PC acquisition devices such as LiDAR, RGB-Depth cameras (e.g., Microsoft Kinect [182], Intel RealSense [183], etc.) and, more recently smartphones (e.g., iPhone 13 Pro [184], Samsung Galaxy S20 [185]) equipped with LiDAR Depth Camera have fuelled its widespread adoption. According to a Straits Research report, the PC market is forecasted to reach 6.93 Billion USD by 2030 [186].

The rising interest in PC has led to increase in PC data availability, thus making it

feasible to leverage deep learning-based techniques such as classification, segmentation, and detection for PC data, thereby facilitating the machines with a better understanding of the scene to make more accurate decisions. Especially, PC deep learning analytics can provide more useful inputs to autonomous cars, drones, robots, etc. by assisting with better visual perception to detect and measure the distance of objects precisely, failing which can result in devastating consequences. Moreover, the availability of LiDAR cameras and neural processing units (NPUs) on the commodity mobile/edge devices have made on-device PC inference an attractive option, thus eliminating the need for offloading computations to server which can potentially incur significant communication latency or energy consumption [187–190].

However, unlike the deep learning analysis on 2D image data, which are structured (where neighboring pixels can be simply found by using indexes), deep learning on PC faces several challenges as the raw PC data are *irregular* (i.e., points are unevenly sampled across different regions) and *unstructured* (i.e., distances between neighboring points are not fixed). Due to these inherent properties of the PC data, two additional steps are employed on inferencing the raw PC data – *sample* and *neighbor search*. However, since the off-the-shelf deep learning acceleration works like [191–194] are not tailored for such operations, the PC analytics applications do not run efficiently on existing hardwares. Our profiling data reveal that just the sample and neighbor search stages contribute to more than 50% of the overall PC inference latency on a typical edge device. So, it is imperative to optimize these two stages, and thereby decrease the latency and energy consumption of PC inference on the edge devices with limited resources and battery life.

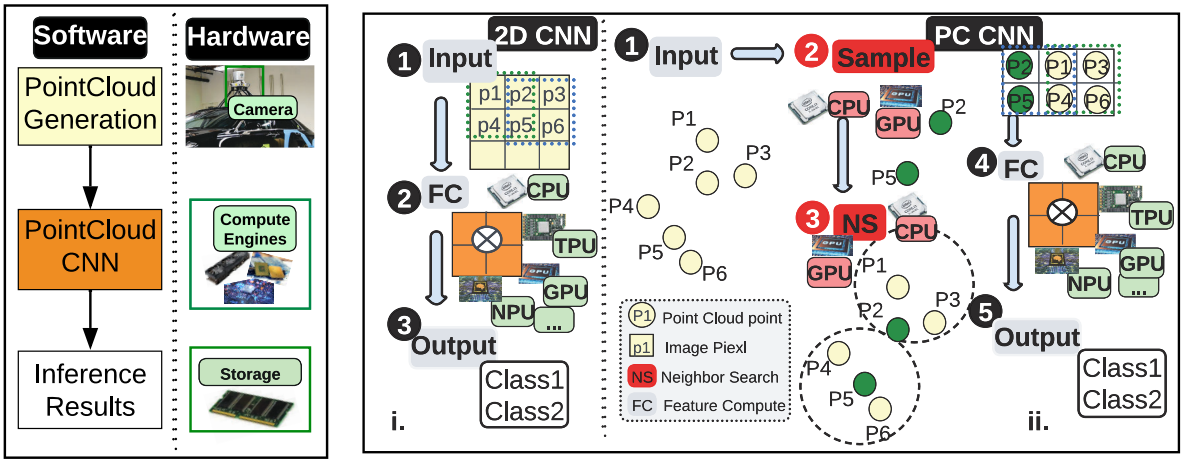
While most prior works along this direction primarily focus on improving the PC inference accuracy [195–198], only few efforts to reduce the inference latency and energy consumption [104,146] on edge devices. Mesorasi [146] accelerated the feature computation stage and pipelined the neighbor search stage to some extent, but did not address the sample stage. Albeit, PointAcc [104] developed custom hardware accelerator for PC inference, but did not consider leveraging potential software-level optimization opportunities by taking into consideration the unique characteristics of the PC data. We want to emphasize that the primary bottlenecks in PC inference – sample and neighbor search stages – are stemmed from the irregular and unstructured nature of the PC data. Hence, driven by insights from 2D image data, where sampling and neighbor search is faster due to its structured representation, "*structurizing*" or "*re-ordering*" the unstructured 3D PC data can provide potential opportunities for accelerating PC inferences.

Morton Code [199], a geometrical data representation method for mapping a multi-dimensional data to one-dimension while keeping the locality among data in tact, can be a potential candidate to achieve such structured representation. However, it is not straightforward to employ Morton Code to speedup the PC inference due to following **challenges**: 1) *how to* ***effectively structurize*** *the PC data using Morton code ?*; 2) *how to* ***efficiently utilize*** *the Morton re-ordered data ?*; and 3) *what are the* ***trade offs*** *between performance benefits, memory overheads as well as the inference accuracy?*

Towards this, in this work, we present and evaluate *EdgePC*, a framework to accelerate the PC inference pipeline as well as improve its energy efficiency on edge devices, with minimal accuracy loss. The key idea is to *leverage* Morton Code to re-arrange the raw points (which are irregular and unstructured) and eventually *maximize* the "structuredness" of point cloud frame. In an ideal scenario, the point cloud frames can be as structured as 2D images. As the re-ordered point cloud frame is more structured, the neighborhood property of the points can be directly inferred by the indexes of each point. In other words, we can treat the re-arranged 3D point cloud similar to a 2D image when performing the CNN inference. Therefore, instead of optimizing the SOTA sampler and neighbor searcher to reduce the execution latency (which have been the focus of prior works [104, 200, 201]), in this paper, we decide to intelligently "skip" these two stages by **approximating** the sampled points as well as their neighbors by simply picking the points with proper indexes from the "structured" point cloud frame, with *minimal* computation. Note however that, such approximation will yield sub-optimal samples and false neighbors (more details in Sec. 6.5), resulting in the accuracy drop in CNN inference. Therefore, to avoid such accuracy loss, we integrate the aforementioned Morton Code-based approximations into the CNN models and retrain the networks.

In summary, our major **contributions** are listed below:

- We first perform a detailed "end-to-end" (E2E) characterization of the typical PC inference application on the state-of-the-art (SOTA) PC analytics workloads and identify the *sample* and *neighbor search* stages to be the primary bottlenecks, due to their inefficient execution on the existing hardware. Furthermore, we demonstrate that this inefficiency stems from the inherent *irregular* and *unstructured* nature of the PC data.
- To address such inefficiencies, we propose a mechanism for using the *Morton Code* to *structurize* the raw PC data. Both our qualitative and quantitative analysis show the advantages of using Morton code in structural data representation. Next, we propose two complementary approaches to efficiently utilize this structured PC data to speedup



(a) Example application.

(b) 2D CNN Vs. PC CNN.

Figure 6.1: (a) PC CNN application example: autonomous driving car; (b) Comparison between 2D CNN and point cloud CNN.

inference: 1) *approximate the sample stage* by uniform sampling on structured PC data, and 2) an *index-based neighbor searcher* to approximate the SOTA neighbor searchers. Finally, to minimize the impact of these approximations on the CNN model precision, we conduct a thorough design space exploration to find an optimal design point to strike the right balance among inference accuracy, performance improvement and memory consumption, which is particularly critical for the limited compute-energy budget edge devices. We also incorporate our approximations for sample and neighbor search stages into the PC CNN models and retrain the weight matrices to minimize the inference accuracy drop.

- We implement our design on an edge GPU development board [202] and evaluate it on six different PC workloads. Our experimental results show that, *EdgePC* can speedup the sample and neighbor search stages by 3.68× which in turn translates to 1.55× speedup of the end-to-end PC inference latency. It also saves 33% of the original overall inference energy consumption with a negligible impact on inference accuracy (compared to the baseline setup).

6.2 Background and Related Work

6.2.1 Background

6.2.1.1 Point Cloud and its Applications

Point cloud (PC), typically captured by the LiDAR cameras, consists of a set of unordered points, with each point associated with a 3D coordinate and its attributes like RGB colors, normals, reflectances, etc. Note that PC is being widely used to represent the 3D models due to its simplicity (it does not require triangle mesh generation to construct surface) and high accuracy (it can preserve the original geometric information [203]). Such representation is essential for many applications like VR/AR [204–206], 3D reconstruction [207–209], autonomous driving [210–212], etc. Recently, deep learning has been utilized in these applications as it can achieve very high accuracy [213]. Fig. 6.1a shows an example to detect objects for autonomous vehicles using a PC-based CNN model. Specifically, the LiDAR camera first scans the surroundings and constructs the PC, following that the underlying compute engine (CPUs, GPUs or NPUs, etc.) performs the CNN inference on the captured PCs. Finally, the detected results (e.g., the cars, pedestrians, trees, etc.) are forwarded to the decision-making engine for the next steps. Such applications not only need high inference accuracy, but also execution efficiency in terms of faster execution for real-time updates, as well as low energy consumption when performed on edge devices. Therefore, the PC CNN inference needs to be carefully designed to meet the above requirements.

6.2.1.2 PC CNN Pipeline

Unlike CNN inference on 2D images, where the inputs are "well-structured" and each pixel and its neighbors can be easily located via "indexes", 3D PCs are essentially unstructured data, and thus, cannot be directly fed into the convolution layers. Given a PC frame, the first step in a PC-based CNN is sampling (obtain a global coverage for the PC frame) and followed by neighbor searching (find local neighbors for each sampled point). For example, as shown in Fig. 6.1b, first, points $P2$ and $P5$ (green color) are sampled from a PC frame. Then, their corresponding neighbors (the points within the dotted circle boundary) are determined. Next, these sampled points and their neighbors ($P2$,$P1$,$P3$; $P5$,$P4$,$P6$) form a 2D matrix (similar to the input shown in Fig. 6.1b for 2D CNN) and then the convolution layers are employed to extract the features. While the feature computations

(FC) (which fundamentally perform matrix multiplications) can be accelerated using the off-the-shelf specialized accelerators like TPUs or NPUs, there are no such customized hardware for sample and neighbor search stages (primary bottlenecks). Thus, minimizing the execution time of these two critical stages and thereby reducing the overall PC CNN execution and energy consumption is the focus of this work.

6.2.2 Related Work

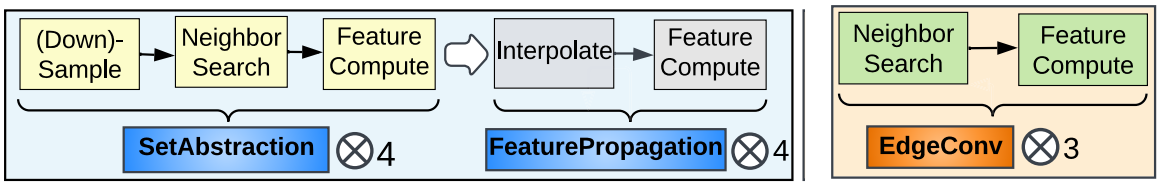
6.2.2.1 PC Analysis

Since Charles et al. proposed the PointNet [214] (the first/leading work that directly handles 3D PC data using deep learning CNNs), deep learning on PCs have been extensively studied in various domains, like 3D shape classification [138, 139, 215], object detection [140–142, 216] and tracking [143, 144, 217] or segmentation [5, 145, 214, 218]. While most of these prior works focus on improving the inference accuracy by designing better CNN architectures, there are also a few targeting to reduce the inference latency and/or improving the energy efficiency, as discussed next.

6.2.2.2 Accelerating PC CNN

To accelerate PC CNN, Mesorasi [146] proposed a delayed-aggregation technique to minimize the feature computation (FC) latency and pipelined the FC and neighbor search stages, while PointAcc [104] proposed a specialized accelerator for mapping and memory management units. In [200], the authors minimized the data movement overheads by increasing the spatial locality of the PC data. Crescent [201] solved the irregular memory accesses in k-d tree-based neighbor search by splitting the tree into top- and bottom-trees. While these techniques can improve the performance of PC CNN to some extent, they have their own limitations. For example, [146] and [201] can only benefit the neighbor search step, but ignore the optimization opportunities for the sample stage, whereas [200] only targets graph-based CNNs, which has a limited application scope. Finally, PointAcc customizes specialized accelerators, which not only involves complex design and tuning cycles, but also misses the potential software-level optimizations, and therefore, could not fully exploit the off-the-shelf hardware such as edge GPUs.

Thus, to the best of our knowledge, there is no prior work investigating the optimization opportunities for the two critical stages in PC pipeline - sample and neighbor search - to minimize the execution time of PC inferences on *edge devices*.



(a) PointNet++ (b) DGCNN

Figure 6.2: Architecture for (a) PointNet++(s) and (b) DGCNN(s).

6.3 Motivation

We first study two widely-used PC CNN pipelines, and then breakdown their latencies to understand their inefficiencies.

6.3.1 PC CNN Latency Characterization

Fig. 6.2 shows the model architectures of two popular PC CNNs used for semantic segmentation tasks – PointNet++ [5] and DGCNN [6]. Primarily, PointNet++ consists of two basic modules, namely, *SetAbstraction* (*SA*) and *FeaturePropagation* (*FP*), with 4 consecutive *SA* modules followed by 4 consecutive *FP* modules. In each *SA* module, the input PC ($N \times C$ matrix, where N is the number of points and C is the feature dimension of each point) is first down-sampled into n points, which can serve as a good coverage of the original input PC. An efficient sampling algorithm for PC CNNs is the farthest point sampling (FPS) [219] , which iteratively selects the farthest point from a set of unsampled points until all the n points are sampled (more details in Sec. 6.5.1). Next, in the SA module, neighbors for each sampled point are searched. Two commonly-used neighbor search methods for this step are ball query [220] and k-nearest neighbor (k-NN) [221]. Finally, the features for these sampled points and their neighbors are grouped into a feature matrix (of dimension $n \times S \times C$, where S is the number of neighbors for each sampled point) which is subsequently fed into the feature computation (FC) stage for feature extraction. The *FP* module can be viewed as *reverse-SA*, where the given n points are first up-sampled/interpolated into N points and then convolved with the convolutional kernels in the FC step. Similar to PointNet++, DGCNN consists of 3 successively connected basic modules (*EdgeConv* (*EC*)), as depicted in Fig. 6.2b. Since the number of points is fixed throughout this network, there is no sampling stage in *EC* (i.e., same number of input points are fed into its subsequent stages).

To better understand the performance implication of these stages, we plot the inference latency of these two CNN models on four datasets using a typical edge SoC

(NVIDIA AGX Xavier [202]) in Fig. 6.3. Specifically, the execution latency is characterized into two main components: sample & neighbor search and feature compute. Overall, across the studied workloads, the sample and neighbor stage can take from 38% to 80% of the end-to-end inference latency. Also, as the number of points increases, these stages take even more time. For example, compared to the ModelNet [222] dataset (with 1024 points/PC), the sample and neighbor search execution latency with the ScanNet [223] dataset (with 8192 points/PC), increases to 80%, making these two stages the primary bottlenecks in the entire inference pipeline. Driven by these observations, we next investigate the reasons behind such inefficiencies, and further explore the potential opportunities for speeding up these stages.

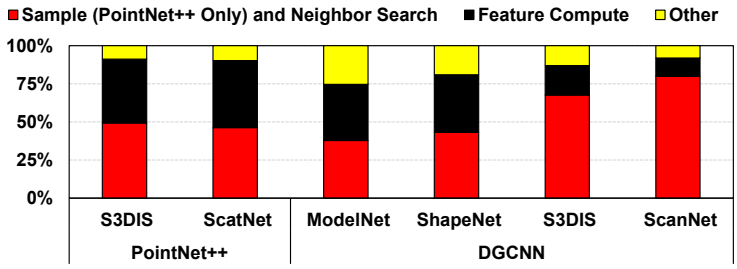


Figure 6.3: Latency breakdown for PointNet++ [5] and DGCNN [6].

6.3.2 Reasons for Inefficiencies and Potential Opportunities

As discussed earlier, the sampling and/or neighbor search stages are the main bottlenecks in the PC CNN inference pipeline. However, considering a 2D image as shown in Fig. 6.4ⓐ-**i**, sampling and neighbor searching can be easily achieved by selecting the pixels with proper indexes with negligible overheads (Fig. 6.4ⓐ-**ii**). Unfortunately, due to the inherent properties (i.e., unstructuredness and randomness) of PCs, simply performing uniform sampling for PC data will result in loss of information. For example, as shown in Fig. 6.4ⓑ-**i**, given a PC frame that contains 12 points (i.e., $\{P1, ..., P12\}$), by the uniform sampling approach, the selected points would be $\{P2, P4, ..., P10, P12\}$ (yellow colored points). As, these sampled points cover only half of the input PC, which is not a good representation of the original PC, this would hurt the CNN models' accuracy. Further, as shown in Fig. 6.4ⓐ-**iii**, for a given pixel (e.g., $p2$) in a 2D image, its neighbors can be found simply via indexing (e.g., $p1, p3, p6$). However, for the point $P2$ in PC (Fig. 6.4ⓑ-**iii**), simply choosing the points with its nearby indices $\{1, 3, 4\}$ would return 2 "false neighbors".

Hence, due to the unstructured nature of PC data, we cannot directly employ the sample and neighbor search techniques used in 2D images (e.g., index-based approaches)

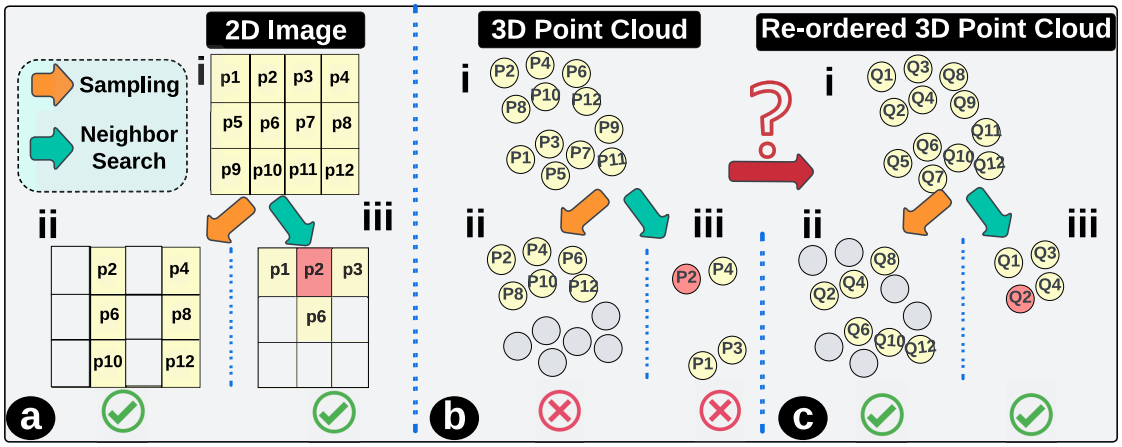


Figure 6.4: Sample and neighbor search for a 2D image (e.g., pixel p2) in (a); 3D PC (e.g., point P2) via indexing in (b); Re-organized 3D PC (e.g., point Q2) via indexing in (c).

for raw PC data. Therefore, 3D PC data uses FPS (for down-sampling), ball query or k-NN (for neighbor search) in order to avoid the loss of important information, at the expense of doubling the execution latency (Fig. 6.3). However, if we can make the PC data more "structured" (a best case would be similar to a 2D image, i.e., 100% structured), then the 2D image sampling and neighbor search techniques can be applied for 3D PC analysis with negligible overheads (e.g., comparing to FPS, ball query or k-NN, etc.), while maintaining an reasonable accuracy. Fig. 6.4**c-i** shows such an example, specifically, after the PCs are re-arranged into a more structured shape, simply performing uniform sampling can return a good coverage for the input PCs (Fig. 6.4**c-ii**). Similarly, the index-based neighbor search accurately determines the neighbors on this structured PC (Fig. 6.4**c-iii**). Compared to the original PC data, this re-organized data is more structured, thus providing the opportunity to utilize the index-based sampling and neighbor search techniques. Note however that, it is *not* trivial to achieve such conversion. Specifically, this conversion technique should meet three requirements: 1) *low complexity* to minimize the overheads; 2) *high parallelism* to fully exploit the existing parallel hardware platforms like GPUs; and 3) *high accuracy* to avoid CNN inference accuracy drop. Fortunately, Morton code, which describes the geometrical relationship between points and can map multidimensional data to one dimension while preserving the spatial locality of the data points, is a perfect candidate to achieve such conversion [78].

In fact, Morton code has been applied to optimize the neighbor search for 3D data in prior works or software libraries. For instance, RTNN [224] has proposed to sort the query points spatially by using the Morton codes (i.e., group the points that are spatially close together), in order to reduce the control flow divergence and better utilize the hardware.

In comparison, works like [225–228] have proposed/implemented grid-based solution strategies for neighbor searching and utilize Morton code to reduce the search space by skipping the searching process for the grids which are far away from the query point. We want to emphasize however that these works focus on "non-approximate" neighborhood search and/or have limited applicability scope. For example, [225] only targets at the ball query-like neighbor search. [229] proposes a $(1+\epsilon)$-approximate nearest neighbor search technique on Morton-sorted points, however, additional computations are required to achieve the specified error bound (i.e., ϵ). On the other hand, as demonstrated later in Sec. 6.5.2.3, our design is an approximation-based solution with no restrictions on the type of neighbor searcher and even further reduces search space/computations and can trade off accuracy for better performance/latency. In other words, while these prior efforts aim to enhance the efficiency for neighbor search with negligible to no accuracy loss, our work prioritizes achieving higher performance with acceptable searching errors, which is more applicable/suitable to PC CNN inference in the context of edge devices.

6.4 Structurizing Point Cloud with Morton code

In this section, we first introduce the Morton code and explain how to generate and utilize Morton code for the PC analysis (Sec. 6.4.1). Next, we demonstrate the effectiveness of applying Morton code for structurizing the PCs using both the qualitative (Sec. 6.4.2) and quantitative (Sec. 6.4.3) results.

6.4.1 What is Morton code?

Morton code (also known as Z-curve [199]), maps data from an n-dimensional space to one dimensional by performing bitwise interleaving on n-d integer coordinates. For example, a point with coordinate $(2,\ 3,\ 4) = (010,\ 011,\ 100)_b$ translates to Morton code $282 = 100,011,010_b$ due to bitwise interleaving. Such mapping can preserve the spatial locality of data points and has been applied in many operations like texture mapping [230], N-body problem [231] and matrix multiplication [232]. However, considering our application (PC), where each point is stored as 3 floating-point coordinates, one critical question is, *how to generate Morton code for non-integer data points?* Simply quantizing the coordinates to the nearest integers might result in information loss. For example, all the points within the $[0, 0.5)\times[0, 0.5)\times[0, 0.5)$ bounds would be quantized as $(0, 0, 0)$. To avoid this, we first divide/voxelize the PC data space (the cuboid of dimension $L\times W\times H$)

into several smaller cubes (SC)/voxels of dimension $r \times r \times r$, where $r < \min(L, W, H)$ is the predefined grid_size. Next, each SC/voxel can be indexed by 3 integers – (i, j, k) where $0 \leq i < L/r$, $0 \leq j < W/r$ and $0 \leq k < H/r$. Given a point, its coordinates can be voxelized into 3 integers depending on which SC/voxel it belongs to, and then the Morton code can be generated by bitwise interleaving. Such transformation from 3-D to 1-D space (3 x, y, z-indexes to 1 Morton code) can simplify the subsequent computations. Due to the spatial locality preservation, the points nearby in space will have similar Morton codes. Thus, to "structurize" the input PC, we *re-order* the points as their Morton codes. Especially, if the original indexes for input points are $I = \{0, ..., N-1\}$ where N is the number of points, after sorting, the new indexes will become $I' = \{i_0, ..., i_{N-1}\}$, where i_0 and i_{N-1} are the indexes of the points with minimum and maximum Morton code values, respectively. The re-arranged points are more structured (e.g., like the pixels in 2D image), and thus, can enable index-based sampling and neighbor searching.

6.4.2 Sampling Structurized Point Cloud

To demonstrate the effectiveness of using "structurized" PC for sampling, in Fig. 6.5, we plot the (down-)sampling results on different PC data (raw PC [7] and "structurized" PC) using two sampling approaches: farthest point sampling (FPS) and uniform sampling. In general, both the FPS on raw data (Fig. 6.5ⓐ) and the uniform sampling on "structurized" PC (Fig. 6.5ⓒ) provide a good coverage for input PC (the sampled points are uniformly distributed across the PC model). While the distribution of the uniformly sampled points on raw PC (Fig. 6.5ⓑ) is either too dense (almost become a continuous line) or too sparse (very few points in certain regions). This "uneven distribution" of points becomes more apparent on further zooming into the PC. On the other hand, although FPS can achieve very good sampling quality, its overhead is too high. Based on our profiling on an edge device (Nvidia AGX Xavier board), sampling 1024 points from the Bunny model [7] (which contains 40256 points) with FPS takes $\approx 81.7ms$, while the uniform sampling consumes only $\approx 1ms$. Clearly, such performance gap between FPS and uniform sampling is expected to further widen when employing larger models (with more points). Fortunately, with the Morton code serving as the bridge to "structurize" the PC data, we can *directly* perform uniform sampling on the "re-ordered" PC data with minimal sampling overhead, while still maintaining a very good quality (similar to the FPS sampling results), as depicted in Fig. 6.5ⓒ.

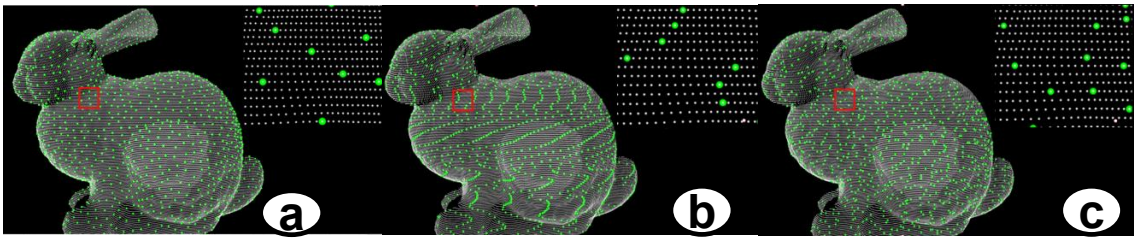


Figure 6.5: Sampled Bunny [7] model via (a) FPS on raw PC; (b) uniform sampling on raw PC; and (c) uniform sampling on sorted/structurized PC data with Morton code.

6.4.3 Neighbor Search for Structurized Point Cloud

Apart from sampling, the other time-consuming stage in PC CNN inference pipeline is the neighbor search stage. This is mainly because of the randomness of PC data, due to which we have to iterate through the whole point set when searching the neighbors for any given point. In this section, we discuss the potential opportunities of "skipping" the neighbor search stage by utilizing the "structured" PC data. For example, given the re-arranged points as well as the new indexes ($I' = \{i_0, ..., i_{N-1}\}$ as discussed in Sec. 6.4.1), we do not need to use complex state-of-the-art (SOTA) neighbor search algorithms (e.g., ball query or k-nearest neighbor (k-NN) which have been widely used in current SOTA PC CNNs). Instead, we can *bypass* the search process by directly selecting k consecutive points near the target data point, where k is the number of neighbors. This means, the k neighbors for the point with index i_p would be the points with indexes $\{i_{p-k/2}, ..., i_p, ..., i_{p+k/2}\}$. By doing so, we can deploy a neighbor search technique similar to the ones used with 2D images (e.g., index-based approach), for 3D PC data as well.

To show the effectiveness of the index-based neighbor searching, Fig. 6.6 plots the *false* neighbor ratio (defined by the ratio of the "neighbors" picked by our above-mentioned scheme but are not identified as neighbors by the SOTA techniques) , where the x-axis is the configurations (i.e., applying different SOTA neighbor search algorithms on different datasets), while the y-axis shows the false neighbor ratio. As we can observe, the false neighbor ratio can be as low as 23%. Furthermore, if we increase the search window (that is, search k neighbors from $\{i_{p-W/2}, ..., i_p, ..., i_{p+W/2}\}$

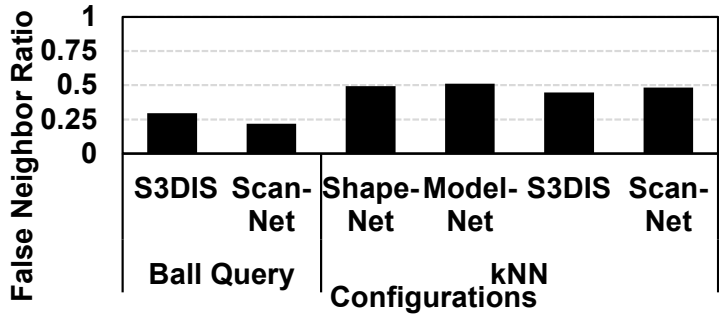


Figure 6.6: False neighbor ratio on different datasets.

instead of directly selecting the $\{i_{p-k/2}, ..., i_p, ..., i_{p+k/2}\}$, where W is the search window size and $W>k$), the false neighbor ratio can be further decreased to 5% (shown later in Sec. 6.6.3).

Takeaways: While such index-based solutions are efficient and fast, they often lead to suboptimal samples and false neighbors. Therefore, simply utilizing the pre-trained PC CNN models will result in decreased inference accuracy; instead, it is required to include the approximations when retraining the models.

6.5 Morton Code-based Sampler and Neighbor Search Design

We observed in Sec. 6.4 that Morton code can be employed to *structurize* the PC data, thus providing the opportunities to use index-based sampling and neighbor searching (the technique used in 2D image). Thus, unlike prior works [104, 201] which try to optimize the sample and/or neighbor search stages by customized accelerators or by choosing specific data structures (e.g., k-d tree, where the tree construction process itself brings non-negligible overheads), in this work, instead of optimizing these two stages, we choose to "*skip*" them by **approximating** the *complex computations on raw PC data* (performed by the SOTA PC CNNs) with the *simple index-based methods on "structured" PC data*.

6.5.1 Morton-code-based Sampler

We first present the SOTA down-sampling technique, farthest point sampling (FPS) [219] and its inefficiencies. We then introduce our proposed design utilizing the Morton code to "structurize" the PC data and our design considerations.

6.5.1.1 Inefficiencies of the SOTA Sampler

To understand the SOTA sampling technique (FPS) for PC data, we illustrate its processing steps in Fig. 6.7 and Fig. 6.8(a). Given a PC data containing N points ($P = \{p_1, ..., p_N\}$) and the desired number of sampled points (n) as inputs, the FPS first initializes the un-sampled(S') and sampled(S) sets as input PC and empty set, respectively, while the elements in array D (which stores the distances between un-sampled points and sampled set) are initialized to be inf. Next, the first sampled point is randomly picked (e.g., s_0), and then both the un-sampled set S' and the sampled

set S, as well as the distance array D are updated accordingly . For the next $n-1$ points, each time a new point is sampled, all the un-sampled points are traversed and the point with maximum distance to the sampled set is picked . Based on the distance array D, the next sampled point can be decided. Note that each time a new point s_i is sampled, the distance array is updated . The time complexity for such update is $O(N)$, and considering that we need to sample n points in total, the time complexity becomes $O(nN) \approx O(N^2)$ (where $n=O(N)$ in practical scenarios). Moreover, all the points in S are inserted sequentially, meaning that there is limited parallelism in FPS, which further adds to the inefficiencies.

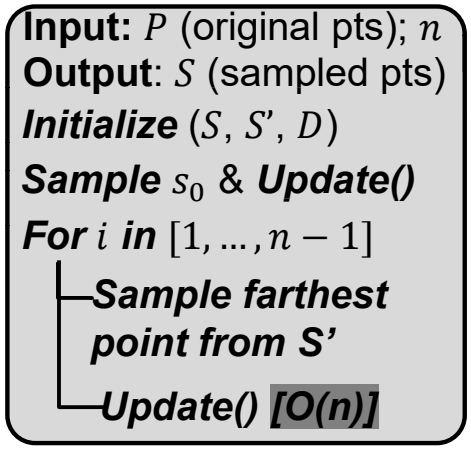


Figure 6.7: Farthest point sampling (SOTA).

Fig. 6.8(a) shows an example using the FPS to sample 3 out of 5 points. First, the sampled and un-sampled sets and the distance array are initialized. Then, $P0$ is sampled, and thus, the (squared) distance array becomes $\{0, 14, 10, 49, 33\}$. Next, the point $P3$ is sampled as its distance to the sampled set is maximum (49). The new distance array is $\{0, 11, 10, 0, 26\}$ and finally the point $P4$ with maximum distance value (26) is sampled.

Takeaway: Although the FPS can yield a good coverage for the input PC data, its compute complexity is too high: $O(N^2)$ for N input points, where N is in orders of 10^6 considering a PC frame with millions of points. Also, due to the data dependency in the sampled set (S) when sampling a new point, all the points must be sampled one-by-one, which limits the performance boosting scope through parallelism. To improve performance, we explore the opportunities by *directly operating on "structured" PC data* (not considered in prior works), and accelerate both the down-sampling and interpolation/up-sampling stages using approximation opportunities.

6.5.1.2 Optimizing the Sampling Stage

Optimizing Down-sampling: Since Morton code can make the PC data more structured, we can perform uniform sampling (with negligible overhead compared to FPS), while still obtain a near-optimal coverage of the input PC data (see Fig. 6.5). Driven by this observation, we design the Morton code-based PC (down-)sampler as shown in Algo. 5

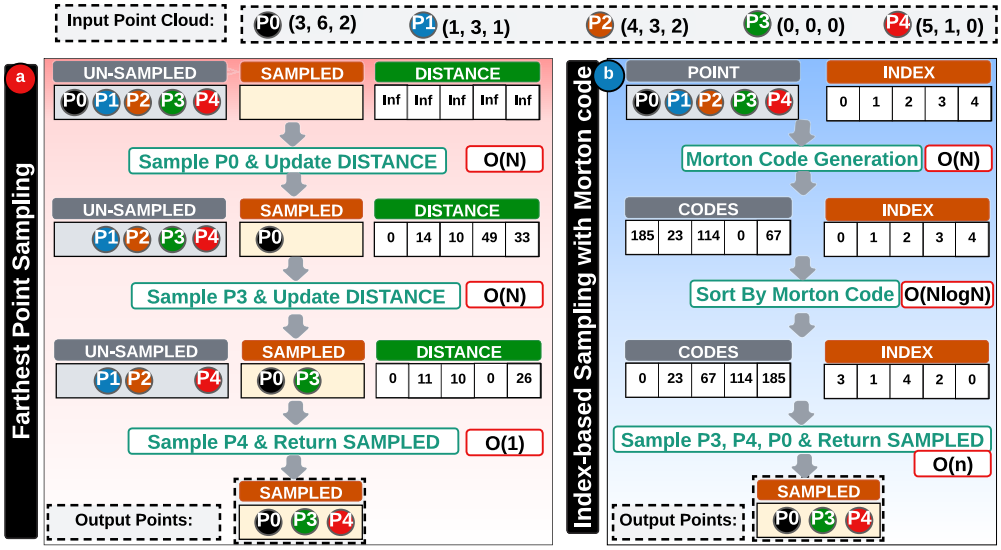


Figure 6.8: Examples: (a) Farthest point sampling on raw PC data; (b) Index-based sampling on Morton code structured PC data.

and Fig. 6.8(b). Overall, our proposal consists of 3 steps, 1) Morton code generation; 2) Morton code sorting; and 3) uniform sampling of the re-ordered/re-organized points. For the Morton codes generation (shown in Algo. 5), we need two more inputs, i.e., r for the predefined grid_size (discussed in Sec. 6.4.1) and $\{x_{min}, y_{min}, z_{min}\}$ array for storing the minimum bounds of PC data. The Morton code generation process can run in a fully-parallel manner, where for each point p_i=(x_i, y_i, z_i), we first calculate which voxel it belongs to (line#4 in Algo. 5), then the Morton code can be obtained by (bitwise) interleaving the indexes for that SC (line#5 in Algo. 5). Due to the high parallelism, generating MC for 8192 points using an edge GPU only takes 0.1ms (0.07% of end-to-end latency). The next step sorts the generated Morton code and outputs the new indexes I'=$[i_0, ..., i_{N-1}]$ in Morton order (line#10 in Algo. 5), where p_{i_0} has minimum the Morton code while $p_{i_{N-1}}$ has the maximum Morton code. The final step uniformly samples the points with the new indexes with a step size of N/n (line#11-13 in Algo. 5). Given a PC frame with N points, the time-complexity of this algorithm is $O(N\log N)$ (due to the sorting stage). Note that, for most cases we have n=$O(N)$ $\gg$log$N$.

Therefore, with our proposal, we can decrease the compute-complexity from quadratic to logarithmic-time. Further, as observed from line#11 in Algo. 5, all the points can be sampled in parallel, which resolves the data dependency issue observed in the FPS technique, as discussed in Sec. 6.5.1.1. Similar to Fig. 6.8(a), in Fig. 6.8(b), we sample 3 points from the input PC (consisting of 5 points). Specifically, given these 5 input points

in this example where $\{x_{min}, y_{min}, z_{min}\}$=$\{0, 0, 0\}$, with a predefined grid_size r=1, we can obtain the Morton codes ($\{185, 23, 114, 0, 67\}$) using the Morton code generation algorithm described in Algo. 5. Sorting these Morton codes outputs the new index array ($\{3, 1, 4, 2, 0\}$), and we can simply perform uniform sampling and pick points P_3, P_4 and P_0, which are exactly the same points (when using the FPS algorithm). However, if we increase the grid_size r, the sampled results might differ from the baseline (FPS) results. For example, if the grid_size is defined as r=4, then the Morton codes would become $\{2, 0, 1, 0, 1\}$, for which the sorted indexes are $\{1, 3, 2, 4, 0\}$ and finally the sampled points are $\{1, 2, 0\}$. In such cases, simply approximating the FPS results with the results of the Morton code-based sampler might degrade the inference accuracy. Thus, we have to carefully decide the grid_size when using the Morton code-based sampler to achieve a good balance between the inference accuracy and the memory overheads for storing the Morton codes (note that a higher grid_size value means having fewer small cubes, and therefore would need fewer bits to represent the indexes for these SCs; more details in Sec. 6.5.1.3).

Algorithm 5: Proposed Morton Code-based Sampler

Input : P=$\{p_0, ..., p_{N-1}\}$:input points; $\{x_{min}, y_{min}, z_{min}\}$
Input : n: number of sampling points; r: preset resolution
Output: $S = \{s_0, ..., s_{n-1}\}$: sampled points

procedure `MC_Gen`(P,r,$\{x_{min}, y_{min}, z_{min}\}$)
  Init: MC= $[0] \times N$
  for p_i ***in*** P **do** `// fully parallel`
    x=(x_i-x_{min})/r; y=(y_i-y_{min})/r; z=(z_i-z_{min})/r
    $MC[i]$=$bitwise\_interleave(x, y, z)$
  returnMC

procedure `MC Sampler`(P,n,r,$\{x_{min}, y_{min}, z_{min}\}$)
  Init: S=$\emptyset$
  MC=`MC_Gen`(P,r,$\{x_{min}, y_{min}, z_{min}\}$)
  $[i_0, i_1, ..., i_{N-1}] = merge\_sort(MC)$
  for k ***in*** $[0, ..., n-1]$ **do** `// fully parallel`
    $index = k \times N/n$; $S = S \cup \{p_{i_{index}}\}$
  return S

Optimizing Up-sampling: As discussed in Sec. 6.3.1, the interpolation/ up-sampling stage in PC CNN (e.g., PointNet++) can be viewed as the "reverse" of the down-sampling stage. Thus, the Morton code-based down-sampling proposal is also applicable for the up-sampling stage. Specifically, the goal of interpolation stage is to "recover" the feature

of the original points ($\{f(p_0), ..., f(p_{N-1})\}$) from the sampled points ($\{f(s_0), ..., f(s_{n-1})\}$). For example, the feature for point p_t can be obtained by $f(p_t)$=$g[f(s_i), f(s_j), f(s_k)]$, where s_i, s_j, and s_k are the 3 closest points to p_t in space where $g[]$ is a predefined function like weighted average. As all the points in S are uniformly sampled from the "structured" PC by picking the points $\{p_{i_0}, p_{i_{step}}, p_{i_{2step}}, ..., p_{i_{(n-1)step}}\}$, where *step* is the step size (N/n). Therefore, for any given point with index i_j (p_{i_j}) where $j \in \{0, ..., N-1\}$, its 4 (approximately) closest points from S should be $\{p_{i_{j'-2step}}, p_{i_{j'-step}}, p_{i_{j'+step}}, p_{i_{j'+2step}}\}$ where j'=j-j%$step$. With this knowledge, we only need to pick 3 closest points out of these 4 points to perform the interpolation, instead of searching through the entire sampled point set S. Therefore, with our Morton code-based up-sampler, the compute-complexity can be decreased by $n/4 = O(n)$.

Takeaway: Our Morton code-based sampler's output can serve as a good approximation for the sampling results obtained from the SOTA, and thus enables fast and efficient uniform sampling for PC data. However, due to such approximation, the sampled points are *sub-optimal* (especially for a large predefined grid_size value), which may degrade the precision of CNN models. Also, as this work targets edge devices with limited resources, we need to carefully configure our designs such that we can achieve a good balance between *performance improvement*, *memory usage*, and *inference accuracy*.

6.5.1.3 Design Considerations

In this subsection, we discuss how our design can consider tradeoffs between the performance improvement, inference accuracy and memory overheads.

Performance improvement vs. inference accuracy: As the proposed Morton code-based sampler can boost the PC CNN performance by enabling uniform sampling and the PC CNNs always have multiple sampling layers (recall from Sec. 6.3.1, PointNet++ [5] consists of 4 down-sampling and 4 up-sampling layers), one may consider to apply such optimization to *all* the sampling layers. However, as observed in Sec. 6.5.1.2, due to the *sub-optimal* sampled results, the precision of CNN model might be degraded. So, instead of applying our Morton code-based approximation to all the sampling layers, we only optimize the critical sampling layer(s), i.e., the ones contributing most to the execution latency. Next, we use Point-

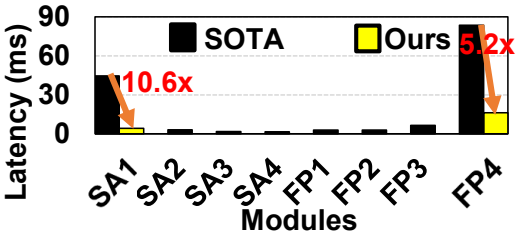


Figure 6.9: Down-sample (in SA modules) and up-sample (in FP modules) latency in PointNet++(s).

Net++ on ScanNet dataset as an example and discuss the details of our design. Fig. 6.9 (black bars) plots the execution latency for all the 8 sampling layers in PointNet++. As can be observed, the down sampling layer in the first SA module and the up-sampling layer in the last FP module are the most time-consuming sampling layers. Therefore, we choose to apply the Morton code-based sampling for these two layers and use the the state-of-the-art samplers for the rest sampling layers. As shown in Fig. 6.9 (yellow bars), we can accelerate the down-sampling layer by 10.6× and the up-sampling/interpolation layer by 5.2×.

Memory overheads vs. inference accuracy: As depicted in Algo. 5, in order to utilize the index-based sampler, first we need to generate the Morton code for each point to structurize the PC data. However, to store these Morton codes, we need to allocate extra space. Considering a PC frame which contains N points, with each point corresponding to a a-bits Morton codes, $Na/8$ Bytes more space will be used. On the other hand, as the PC data contains the 3D coordinate information, these a bits need to be further split into 3 parts ($\lfloor a/3 \rfloor$ bits for each dimension), meaning that the entire space can be divided into $2^{\lfloor a/3 \rfloor} \times 2^{\lfloor a/3 \rfloor} \times 2^{\lfloor a/3 \rfloor}$ small cubes (SCs), which will further translate to a grid_size value of $r{=}D/2^{\lfloor a/3 \rfloor}$, where D is the dimension of the PC's bounding box. Intuitively, the inference accuracy of the CNN models can benefit from a larger a (corresponding to a smaller r), at the expense of higher memory overheads, and vice verse. In this work, we choose $a{=}32$ to achieve a good balance between the memory overhead and the inference accuracy degradation.

6.5.2 Morton Code-based Neighbor Search

In the previous section, we have explored the opportunities of using Morton code to structurize the raw PC data and thus boost the performance for sample stages. Recall from Sec. 6.4.3 that, the neighbor search stage can also benefit from the "structured" PC data. Before diving into the details of our proposed optimizations, let us first understand the state-of-the-art neighbor search approaches and their inefficiencies.

6.5.2.1 Inefficiencies of the SOTA Neighbor Search

Ball-Query (BQ) [220] and k-NN [221] are two of the most commonly used neighbor searchers. Specifically, given a point p_i and the number of neighbors to be searched (e.g., k), the BQ algorithm searches through the candidate points set (e.g., $P{=}\{p_0, ..., p_{N-1}\}$) and returns the points inside the ball with center p_i and a predefined radius R. To achieve

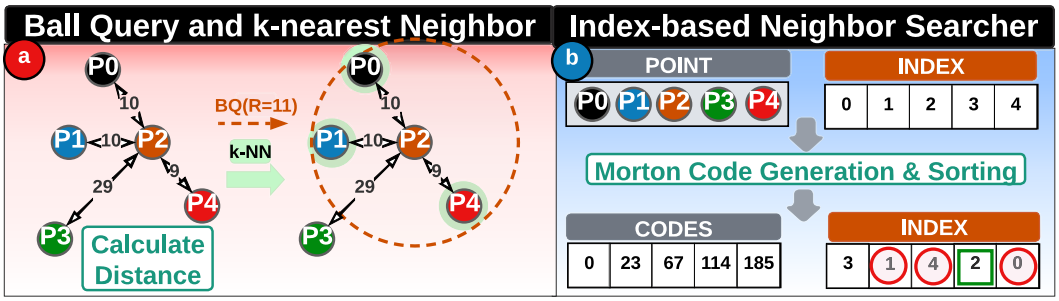


Figure 6.10: Examples for (a) ball-query and kNN; (b) index-based neighbor search with Morton code structured PC data.

this, the distance between p_i and any other points ($dist(p_i, p_j)$) for any j in $\{0, ..., N\text{-}1\}$ needs to be computed in $O(N)$ time. Thus, searching the neighbors for all the N points would take $O(N^2)$ time. Fig. 6.10(a) shows an example using the BQ algorithm to search 3 neighbors for $P2$, which takes the same PC data in Fig. 6.8 as input, and defines the radius (R) as 11. Following the above-mentioned flow, first, it computes the distance between $P2$ and the other points. Next, $P0$, $P1$, and $P4$ are picked as their distance to $P2$ is less than R. Similarly for k-NN, the first step is also obtaining the distance matrix (with a time complexity of $O(N^2)$[1]; next, the 3 points with green background are selected as they have minimum distances.

Takeaway: As the PC data contains up to millions of points, any algorithm with $O(N^2)$ complexity would be too costly for edge devices. So, an efficient neighbor searcher needs to be designed to reduce the computation complexity, while still maintaining a high parallelism and inference accuracy.

6.5.2.2 Optimizing the Neighbor Search Stage

Motivated by the index-based neighbor searcher used for 2D images and by the observation that Morton code can be used to structurize the PC data from Sec. 6.4 and Sec. 6.5.1.2, in this section, we propose to *approximate* the SOTA neighbor search process by simply selecting/searching the points whose indexes are near the target point (the point for which we search the neighbors) in the "structured" PC data. For example, after we obtain the new index array (I'=$\{i_0, ..., i_{N-1}\}$, by sorting the Morton codes, as discussed in Sec. 6.5.1.2), for any given point p_{i_j}, instead of searching its neighbors from the entire space (which is the case in SOTA works), now we only need to search through the points

[1]Although prior ball-query or k-NN implementations – like the kd-tree-based approaches, have lower complexity ($O(NlogN)$), both tree construction and traversal have limited parallelism, and hence, are inefficient.

with indexes of $\{i_{j-W/2}, ..., i_{j+W/2}\}$ ($k \leq W \leq N$ is a predefined *search window size*), by which the time complexity is reduced from $O(N)$ to $O(W)$. Note that, W is normally set to be much smaller than N, meaning that the performance improvement brought by our proposal is expected to be high. In the example shown in Fig. 6.10(b), W is defined as k+1 (4=3+1), within which $P1$, $P4$, and $P0$ are selected.

Takeaway: Our index-based neighbor searcher on "structured" PC data can reduce the time complexity of the neighbor search stage by $O(N/W) \approx O(N)$, thus improving the performance. However, similar to the issue of the proposed sampler discussed before, since the the Morton code cannot make the PC data 100% structured, the searched neighbors using this proposal is also *sub-optimal* and might contain "false neighbors". This can potentially lead to quality degradation of CNN models. Therefore, in next section, we explain our design considerations on *how to maximize the performance improvement, while minimizing the impact (caused by the proposed approximations) on CNN models.*

6.5.2.3 Design Considerations

Similarly, the first question we want to ask here is: *for which layer(s) should we apply our approximations?* Recall that, PointNet++ [5] has 4 neighbor search layers. We plot the performance improvement and the ratio of false neighbors for each layer in Fig. 6.11 to show the impact of our proposal on these layers. As observed, layer1 has the most significant speedup and the least false neighbor ratio. As a result, it is a perfect candidate for applying our optimizations. Moreover, since we also apply the Morton code-based approximations for the first sample layer (Sec. 6.5.1.3), and the neighbor search is right after the sample stage, we can simply reuse the Morton code for our neighbor searcher without any extra overhead. However, for the rest layers, we can only obtain limited performance boost at the expense of much larger false neighbor ratios, which may further result in CNN model degradation. Also, generating the Morton codes for these layers would introduce extra overheads and further decrease the benefits of our proposal.

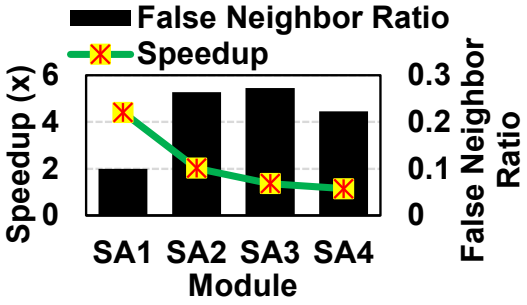


Figure 6.11: Our neighbor searcher speedup vs. false neighbor ratio for 4 modules in PointNet++(s).

Therefore, similar to the design for sampler discussed in Sec. 6.5.1.2, we only apply the optimization for the first neighbor search layer. As opposed to PointNet++ where all

the modules use the x, y, z coordinates of input PC for neighbor searching, in DGCNN [6], only first module uses the geometry coordinates for determining the nearest neighbors. For the next several modules, the distance between points are measured using the features (i.e., $dist(p_i, p_j)$=$dist(f_i, f_j)$ where f_i is the feature vector of p_i and always has higher dimensionality (e.g., 64)). Due to this, our Morton code-based optimization can only be applied for the first module as it cannot process higher-dimensional data. For the following neighbor search layers, instead of performing the SOTA computations, we decide to interleave the "reuse" and "compute". For example, with the reuse distance of 1, we reuse the neighbor search results from layer1 for layer2, and employ the SOTA algorithm for layer3. The intuition behind this is that, during the propagation the the CNN model, the neighborhood of points would not vary much across consecutive layers.

Note that, both the reused Morton codes and search results are stored in the GPU memory. As the PC data is processed in batches (shown later in Table. 6.1), the per-batch size of the Morton code and reused search data are only up to 32KB and 160KB, respectively, leading to reduced memory access overheads. Further, our optimization reduces the neighbor search computations, resulting in fewer GPU memory accesses and also reducing the overall data request stalls by 4-5%.

After deciding the layer where approximation will be applied, the next question is: *how to decide the search window size*, which *is the key parameter to trade off between execution latency and model precision?* In fact, with our proposal, the user can adaptively select proper search window size to accommodate the application requirement. We further investigate how different search window sizes shape the behavior of our approach in a sensitivity study in Sec. 6.6.3.

6.5.3 What are the Pros and Cons of Approximation?

As discussed in Sec. 6.5.1.2 and 6.5.2.2, compared to the SOTA sampling and neighbor search techniques, our proposal first uses the Morton codes to structurize the unordered PC data, thereby enabling the index-based operations. This is expected to be much faster. In fact, we show later in Sec. 6.6.2, with our proposal, the sampling and neighbor search can achieve up to 5.2× speedup. However, as discussed earlier, due to the suboptimality of the sampled points and the existence of false neighbors as well as the reuse of neighbor indexes across CNN modules, the inference precision of CNN models is expected to be degraded. To address this issue, we must retrain the CNN models taking into consideration the Morton code-based approximation (discussed further in Sec. 6.6.2).

6.5.4 Architectural Insights – Optimizing the Shifted Bottlenecks

Our Morton code-based approximation techniques can accelerate the sample and neighbor search stages by 3.68x(shown in Sec. 6.6.2). In this section, we explore further opportunities to improve performance/energy efficiency and provide some architectural insights as future research directions.

6.5.4.1 Increase Tensor Core Utilization for Feature Computation

Recall from the execution latency breakdown in Fig. 6.3 that, after our optimizations, now the main bottleneck for PC inference shifts to the feature compute (convolution) stage. Fortunately, as recent edge devices are equipped with the tensor cores [233], which are customized for accelerating matrix multiplications, as shown later in Sec. 6.6.2, by employing the tensor cores, the inference can be further accelerated by 27%. However, we observe that the tensor core utilization is basically *zero* for several layers. This is mainly because the channel dimension of the input matrix is below a certain threshold, and the tensor cores are not invoked. In fact, based on our profiling, with an input matrix of dimension 32×1000×12×32 and the weight matrix of dimension of 12×64×1×1, the convolutional layer takes 40.4ms to execute with no tensor core utilization. However, if we resize the input matrix into 32×100×120×32 shape and convolve it with a 120×64×1×1 dimensional weight matrix, although the compute complexity remains the same, the execution latency reduces to 18.3ms, with a 40% tensor core utilization. Driven by this observation, one optimization direction might be *extending the input feature dimension by gathering the features of several nearby points.* For example, given an input matrix of NxkxC where k is the number of neighbors for each of the N points, and C is the original feature dimension, instead of performing the feature computation for each point P_i, we can first merge the features of several (t) nearby points (like $[f_{it},...,f_{(i+1)t-1}]$, by which the feature dimension is increased to Ct) and then perform the FC. Finally, we can split the convolution result back for these t points (e.g., by averaging). With our Morton code based reordering, the points which are nearby in the input matrix are also close to each other in the space. Therefore, such approximation (merge and split) is not expected to degrade the model quality much. To make this work, the data layout and the approximation techniques have to be carefully designed to avoid any memory access overheads and maintain the CNN model precision.

6.5.4.2 Decreasing the Data Movement Overheads in Grouping

Another time-consuming operation in the optimized PC CNN pipeline is the *grouping*, which gathers the features of each sampled point and that of their neighbors to form a new feature matrix. For example, given an input feature map of shape NxC and a neighbor index matrix I_M of shape nxk, where N and n are the number of points and the number of sampled points, respectively, while C is feature dimension of each point, after the grouping stage, the dimension of new feature matrix is nkxC. In practice, nk is larger than N (e.g., for PointNet++, nk=8N), assuming that each GPU thread gathers the feature for one index (out of nk in total), there must be some threads reading *exactly* the same data from memory. Driven by this observation, we want to ask, *can we properly schedule the GPU threads such that there are data sharing/reuse opportunities across them?* In fact, based on our profiling, with simply sorting the index matrix (after which the indexes in each row of I_M is in ascending order), the amount of data transferred/read from L2 cache and system memory can be decreased by 53.9% and 25.7%, respectively. These initial results are encouraging to pursue Morton code-based architectural design space exploration in future.

6.6 Experimental Evaluation

We implement and evaluate our proposals against the SOTA PC inference across three metrics critical for PC CNN application: execution latency (for both sampling and neighbor search stages as well as the E2E inference pipeline), energy consumption, and model precision (accuracy). We first describe our experimental platform, PC CNN workloads (the CNN models and the datasets), and the design configurations (Sec. 6.6.1). Next, we analyze the results (Sec. 6.6.2) and vary the design points to present the sensitivity study (Sec. 6.6.3). Finally, we compare our proposal to prior works (Sec. 6.6.4).

6.6.1 Methodology

6.6.1.1 Experimental Platform

We use NVIDIA Jetson AGX Xavier [202], an edge development board (Fig. 6.12), which consists of a 512-core Volta GPU with 64 Tensor cores, a 8-core ARMv8 64-bit CPU, and 16GB LPDDR4x memory.

Table 6.1: Workloads used in this work.

Workload	Model	Dataset	#Points/Batch	Task
W1	PointNet++(s) [5]	S3DIS [234]	8192	Semantic Segmentation
W2		ScanNet [223]	8192	
W3	DGCNN(c) [6]	ModelNet40 [222]	1024	Classification
W4	DGCNN(p) [6]	ShapeNet [235]	2048	Part Segmentation
W5	DGCNN(s) [6]	S3DIS [234]	4096	Semantic Segmentation
W6		ScanNet [223]	8192	

6.6.1.2 Workloads

To demonstrate the effectiveness of our proposals, we use two popular PC CNNs, namely, PointNet++ [5] and DGCNN [6] and four datasets as listed in Table. 6.1. Specifically, both S3DIS [234] and ScanNet [223] are indoor scene datasets, on which the PointNet++(s) [5] and DGCNN(s) [6] are employed to perform the semantic segmentation. ModelNet40 [222] and ShapeNet [235] are two synthetic datasets. We use DGCNN(c) and DGCNN(p) to perform classification and part segmentation on these two datasets, respectively. All PC frames in these datasets are preprocessed into several mini-batches, with each batch having a fixed number of points, as shown in Table 6.1.

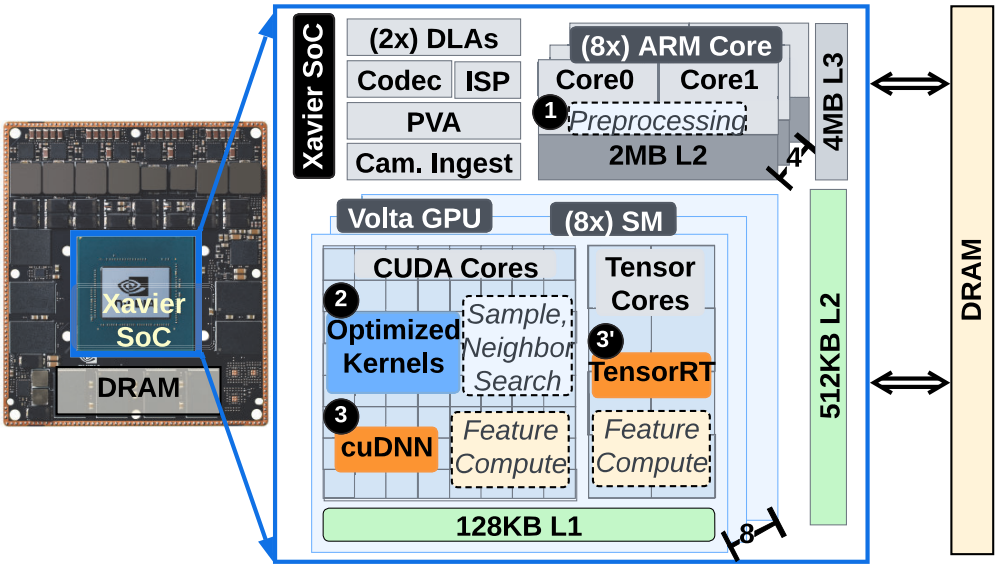
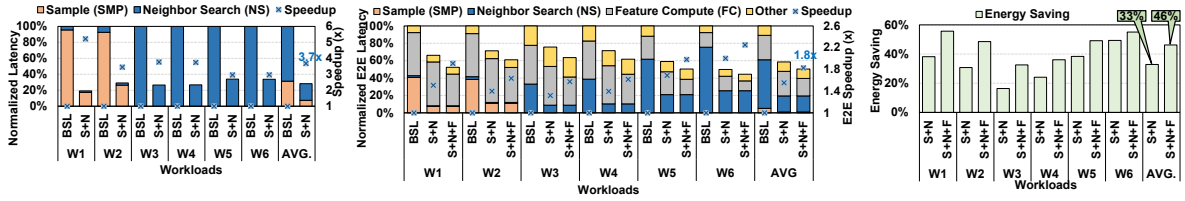


Figure 6.12: Experimental setup.



(a) Speedup (Sample and Neighbor Search) (b) Speedup (end-to-end, E2E) (c) Energy Saving

Figure 6.13: Performance of six workloads w.r.t. the baseline: (a) sample and neighbor search, and (b) E2E speedup; (c) energy saving.

6.6.1.3 Configurations

Baseline: We evaluate the baseline inference on the Volta edge GPU and implement the CNN inference using PyTorch [75] with cuDNN (for feature compute stage). The sampling and neighbor search stages are optimized using CUDA kernels.

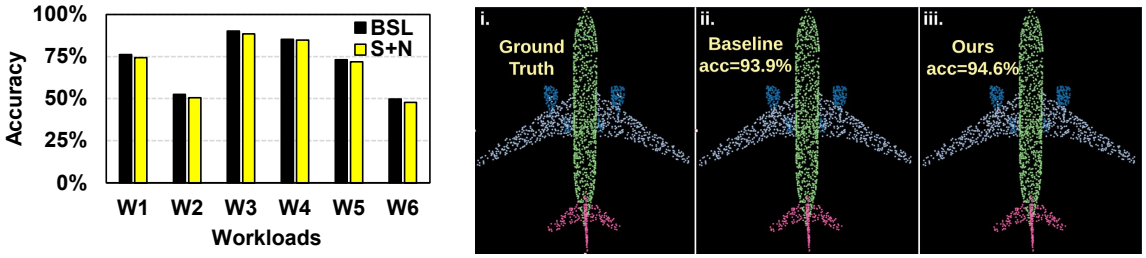
S+N: We implement the proposed Morton code-based approximation techniques with custom CUDA kernels and apply them to both **S**ample and **N**eighbor search stages (step-❷ in Fig. 6.12). We use 32 bits for the Morton code because based on our sensitivity study, as the number of bits required to store Morton code increase, the false neighbor percentage (in neighbor search stage) reduces till 32 bits and further increasing the bits does not yield much benefit.

S+N+F: To further boost the end-to-end inference pipeline, we deploy the **F**eature compute (FC) stage to Tensor cores available on the Volta GPU (step-❸' in Fig. 6.12).

6.6.2 Performance Results

We present and compare the normalized execution latency (left axis) and speedup (right axis)of our proposals w.r.t. to baseline and energy consumption (via the built-in *tegrastats* [236] tool on the Jetson board) for each workload, as well as the accuracy to evaluate our proposals' effectiveness, and plot the experimental results in Fig. 6.13 and observe the following:

Execution Latency: Overall, our optimizations can accelerate the sampling (SMP) and neighbor search (NS) stages by 3.7×, on average, w.r.t. the baseline, as shown in Fig. 6.13a. Across the workloads using the PointNet++ model, W1 can benefit more from our proposals compared to W2 (e.g., 5.21× vs. 3.44× speedup). This is mainly because the inference is performed at batch-level as mentioned in Sec. 6.6.1.2, and the batch size



(a) Accuracy comparison (b) Ground truth vs baseline vs ours

Figure 6.14: (a). Accuracy; (b). Demo: part segmentation with the baseline inference and our proposal.

of W1 is fixed (32) and is usually larger than that of W2 (ranging from 4 to 41 depending on the PC frame, with an average batch size of 14). Recall that, as mentioned in Sec. 6.5, the baseline sampler and neighbor searcher have quadratic-time compute complexity, as a result, each batch is processed sequentially due to lack of available compute resources on the resource-constrained edge device. As a result, the execution time for the SMP and NS stages increases from about 33ms/batch for ScanNet to 76ms/batch for S3DIS. On the other hand, as our proposal can decrease the computational complexity of the down- and up-sampling stages by $O(N/logN)$ and $O(N)$, respectively, and thus enabling to process multiple batches in parallel, the gap between the execution latency for the SMP and NS stages on ScanNet and S3DIS reduces to only 4.9ms/batch (it takes 9.7ms/batch for ScanNet and 14.6ms/batch for S3DIS), which is the main reason for the performance gains observed in W1 and W2. The speedup for the rest of the workloads with DGCNN are ≈3 or 4. Specifically, for both W3 and W4, our proposal can achieve around 3.7× speedup. Such acceleration comes from two sources: 1) due to the approximation of the SOTA neighbor searcher with our proposal for first *EC* module (described in Sec. 6.3), the NS stage can get 29× (W3) and 14.2× (W4) speedup (due to the reduction in time complexity as explained in Sec. 6.5.2.3); 2) owing to the reuse of the neighbor indexes, the NS computation can be skipped for the second and fourth *EC* modules. A similar trend can be observed in W5 and W6 as well. The performance improvement for the SMP and NS stages can further translate to a 1.55× average speedup in terms of the end-to-end (E2E) latency as shown in Fig. 6.13b. Moreover, when the tensor cores are employed for accelerating the FC stage, the E2E inference can be further accelerated by up to 2.25× (W6).

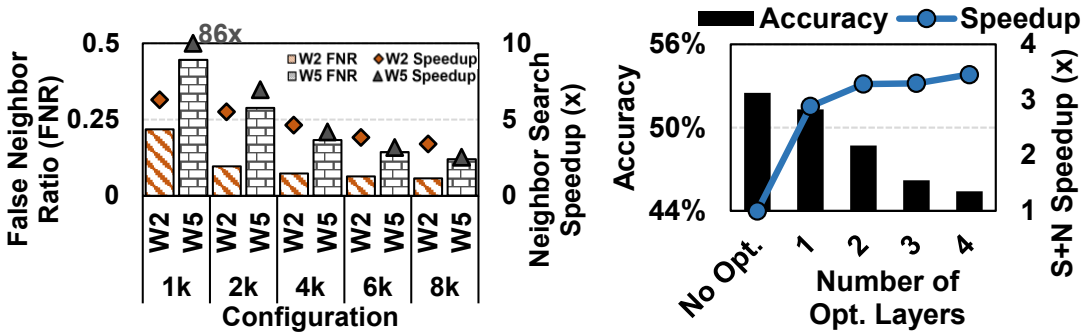
Energy Consumption: Furthermore, the computation reduction for the SMP and NS stages when using our Morton code-based schemes can reduce the energy consumption

as well. As shown in Fig. 6.13c, with our design, the energy consumption for inferencing one PC frame is decreased by 33% on average, and 13% more energy can be saved by deploying the feature compute stage (comprising of matrix multiplication operations) into the tensor cores (instead of just using the CUDA cores). For W1 and W2 which employ the PointNet++, our proposal can achieve 38% and 31% energy savings, respectively. Apart from the reduced execution latency as discussed above, another reason for such savings come from a lower power consumption when applying our approximation techniques (e.g., power consumption decreased from $4.5W$ to $4.2W$ for W1). For the remaining four workloads, however, the energy savings are slightly lower compared to their corresponding execution latency reduction. For example, the 1.32× speedup observed with W3 (Fig. 6.13b), only translates to 16% energy savings (Fig. 6.13c). This is due to our reuse proposal which induces an increased memory pressure when the neighbor index array from the previous *EC* module is cached/stored for later reuse. Based on our measurements, the power consumption of the memory increases from $1.35W$ for the baseline setting to $1.63W$ in our proposal.

Accuracy: Due to the suboptimality of our Morton code-based sampler and neighbor searcher, directly using the pretrained CNN models to perform the inference can cause an accuracy drop. So, to maintain the accuracy, we need to retrain the CNN models with our proposed approximations. Fig. 6.14a shows the accuracy comparison of our retrained models with the baseline. As observed, the accuracy drop is within 2%, thereby having a minimal impact on the inference quality, as shown in Fig. 6.14b. Moreover, for the accuracy-sensitive applications, we can opt to trade the performance for accuracy. For example, for workload W3, with a larger search window size, our accuracy drop can be as low as 0.7% while simultaneously achieving 4.2× speedup. This reflects the "flexibility" of our proposal in adapting to different application-level requirements and execution environment constraints (accuracy- vs latency-sensitive).

6.6.3 Sensitivity Study

We study the sensitivity of our proposal by varying the design points discussed in Sec. 6.5. Fig. 6.15a shows the tradeoffs between false neighbor ratio (FNR) and speedup for the neighbor search (NS) stage when varying the search window size (where k is the number of neighbors), thus providing insights into efficiently choosing the proper search window size. For example, the accuracy-sensitive applications can use a larger search window for preserving the accuracy, while the high throughput demanding applications can prefer a smaller search window to boost the performance. Another design consideration



(a) False neighbor ratio vs search window size vs neighbor search speedup.

(b) Accuracy vs number of optimization layers vs speedup.

Figure 6.15: Sensitivity study.

is the number of layers/modules to apply our MC-based optimization. As shown in Fig. 6.15b, with only the first SA module (and the corresponding FP module) being optimized, the sampling and neighbor search stages can be sped up by 2.9× with only 1.2% accuracy drop. However, when we apply the MC-based approximation to more layers, the performance only improves slightly with the significant accuracy drop. This demonstrates the effectiveness of our proposal. To summarize, given new workloads, the developer can first perform the characterization (like the one in Sec. 6.3) to identify the bottleneck layer(s), for which the MC-based optimizations will be applied and the parameters (e.g., search window size) can be adaptively chosen to accommodate the application's requirement.

6.6.4 Comparison against Prior Works

We compare our proposal with the prior works from both the quantitative and qualitative perspectives. We first implemented the delay-aggregation (DA) technique in [146] for PointNet++ [5] and tested on S3DIS [234]. The feature compute (FC) stage can be accelerated by 2.1× with DA (88.2ms to 42.2ms per batch). However, as also depicted in [146], since the feature dimension usually increases after convolutional layers, the aggregation/feature grouping stage will become the primary bottleneck. For example, after applying DA, the latency for feature grouping stage increases by 2.73×. Further, as it has no optimization for sampling stage, DA can only achieve 1.12× speedup for the E2E inference latency. PointAcc [104] is another work which optimize the PC CNN by customizing specific accelerators (e.g., mapping unit and memory management unit). Note however that, *our work is orthogonal to PointAcc.* Especially, the key module in the mapping unit in [104] is the distance calculation (with $O(N^2)$ time

Table 6.2: Qualitative comparisons.

	Accuracy	Generality	Design Overhead
Crescent [201]	✓	✓	✗
PointAcc [104]	✓	✓	✗
Point-X [200]	✓	✗	✗
EdgePC	✓	✓	✓

complexity). With our work, we only need to calculate the Morton codes for each point (with $O(N)$ time complexity), meaning that, by deploying our proposal in the mapping unit in PointAcc [104], the performance of PC CNN inference can be further boosted. Point-X [200] increases the energy efficiency for graph-based PC CNNs by extracting the spatial locality via a SBFS graph traversal algorithm. However, we want to emphasize that, compared to graph traversal, Morton code is a better candidate for capturing the spatial locality of PC data due to its simplicity and has also been proven to be efficient in a recent work [78]. Furthermore, Point-X [200] has very limited application scope and lacks generality as shown in Table. 6.2 since it is only targets the graph-based PC CNNs. Recently, [201] addresses the irregular memory access issue in neighbor search stage by splitting the k-d tree into top- and bottom-trees. This work, however, overlooks the sampling stage. Finally, all these prior works introduce extra design overheads for hardware customization, while EdgePC favors the commercially available hardwares and can boost the performance without any overheads.

6.7 Conclusion

Point Cloud (PC) has recently gained huge attention with the increasing availability of low-cost PC acquisition devices like smartphones with LiDAR cameras. In particular, the availability of specialized accelerators (e.g, NPU, TPU) on smartphones/handhelds have made PC inference on edge devices an attractive option. In 3D PC deep learning analytics, sampling and neighbor search stages contribute to more than 50% of the end-to-end inference latency, thus are the primary bottlenecks in the entire PC inference pipeline. Although few prior works have either designed custom PC accelerators in hardware or proposed software optimizations, they have missed opportunities to optimize these two critical stages, especially considering the unique characteristics of the PC data (irregular and unstructured). In this paper, we present a novel technique to utilize *Morton code* to "structurize" the raw PC data and minimize the sampling and neighbor search latencies by

intelligently skipping computations on the structured PC data. Towards this, we design *EdgePC*, which is an edge-friendly framework with two complementary approximation techniques for the sampling and neighbor search stages in order to accelerate the PC inference as well as improve its energy efficiency. Our evaluations of six different PC workloads on an NVIDIA Jetson Xavier edge board demonstrate that the proposed design does not only achieve an average speedup of 3.68× for the sampling and neighbor search stages (which translates to 1.55× speedup of the end-to-end PC inference latency) with minimal accuracy loss, but it also saves 33% of the overall PC inference energy.

Chapter 7
Conclusions and Future Work

Bibliography

[1] Xu, M., M. Zhu, Y. Liu, F. X. Lin, and X. Liu (2018) "DeepCache: Principled Cache for Mobile Deep Vision," in *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, p. 129–144.

[2] Kitware, Inc. (2011), "The VIRAT Video Dataset," "https://viratdata.org".

[3] MPEG (2022), "Point Cloud Video: Redandblack," "https://bit.ly/3NyQq2R".

[4] ——— (2022), "Point Cloud Video: Loot," "https://bit.ly/3QXDsPf".

[5] Qi, C. R., L. Yi, H. Su, and L. J. Guibas (2017) "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Curran Associates Inc., Red Hook, NY, USA, p. 5105–5114.

[6] Zhang, M., Z. Cui, M. Neumann, and Y. Chen (2018) "An End-to-End Deep Learning Architecture for Graph Classification," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI Press.

[7] Stanford University Computer Graphics Laboratory (1994), "The Stanford Models," "http://graphics.stanford.edu/data/3Dscanrep/".

[8] Eugene d'Eon, Bob Harrison, Taos Myers and Philip A. Chou (2017), "JPEG Pleno Database: 8i Voxelized Full Bodies (8iVFB v2) - A Dynamic Voxelized Point Cloud Dataset," "https://bit.ly/3cJQ61a".

[9] Charles Loop, Qin Cai, Sergio Orts Escolano, and Philip A Chou, "JPEG Pleno Database: Microsoft Voxelized Upper Bodies - A Voxelized Point Cloud Dataset," "http://plenodb.jpeg.org/pc/microsoft".

[10] Pateşan, M., A. Balagiu, and C. Alibec (2018) "Visual aids in language education," in *International Conference Knowledge-Based Organization*, vol. 24, pp. 356–361.

[11] eLearning Blogger, S. (2023), "Studies Confirm the Power of Visuals to Engage Your Audience in eLearning," https://www.shiftelearning.com/blog/bid/350326/studies-confirm-the-power-of-visuals-in-elearning.

[12] Raiyn, J. (2016) "The Role of Visual Learning in Improving Students' High-Order Thinking Skills." *Journal of Education and Practice*, **7**(24), pp. 115–121.

[13] Budzinski, O., S. Gaenssle, and N. Lindstädt-Dreusicke (2021) "The battle of YouTube, TV and Netflix: an empirical analysis of competition in audiovisual media markets," *SN Business & Economics*, **1**(9), p. 116.

[14] Hirose, A. (2022), "2023 Guide to Creating Stunning Visual Content for Social Media," https://blog.hootsuite.com/epic-guide-creating-social-media-visuals/, accessed: December 22, 2023.

[15] Du, M. and X. Yuan (2021) "A survey of competitive sports data visualization and visual analysis," *Journal of Visualization*, **24**, pp. 47–67.

[16] Craig, E. and M. Georgieva (2018) "VR and AR: The Art of Immersive Storytelling and Journalism," *EDUCAUSE Review*.

[17] Nvidia Corporation (2018), "Jetson AGX Xavier Developer Kit," "https://bit.ly/3oWQNtH".

[18] Yang, T.-J., A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam (2018) "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 289–304.

[19] Niu, W., X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren (2020) "PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 907–922.

[20] Ma, X., F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang (2019) "PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-time Execution on Mobile Devices," *arXiv preprint arXiv:1909.05073*, pp. 5117–5124.

[21] Courbariaux, M., I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio (2016) "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*.

[22] Courbariaux, M., Y. Bengio, and J.-P. David (2015) "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, p. 3123–3131.

[23] Zhou, A., A. Yao, K. Wang, and Y. Chen (2018) "Explicit Loss-Error-Aware Quantization for Low-Bit Deep Neural Networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9426–9435.

[24] Leng, C., H. Li, S. Zhu, and R. Jin (2017) "Extremely Low Bit Neural Network: Squeeze the Last Bit Out with Admm," *arXiv preprint arXiv:1707.09870*, pp. 3466–3473.

[25] Wu, J., C. Leng, Y. Wang, Q. Hu, and J. Cheng (2016) "Quantized Convolutional Neural Networks for Mobile Devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4820–4828.

[26] Chen, T., T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy (2018) "TVM: End-to-End Optimization Stack for Deep Learning," *CoRR*, pp. 578–594.

[27] Tencent (2017), "NCNN," `"https://github.com/Tencent/ncnn"`.

[28] ——— (2020), "TNN," `"https://github.com/Tencent/TNN"`.

[29] Alibaba (2019), "MNN: Mobile Neural Network," `"https://github.com/alibaba/MNN"`.

[30] Zhu, Y., A. Samajdar, M. Mattina, and P. Whatmough (2018) "Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, p. 547–560.

[31] Riera, M., J.-M. Arnau, and A. González (2018) "Computation Reuse in DNNs by Exploiting Input Similarity," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, p. 57–68.

[32] Mazumdar, A., B. Haynes, M. Balazinska, L. Ceze, A. Cheung, and M. Oskin (2019) "Perceptual Compression for Video Storage and Processing Systems," in *Proceedings of the ACM Symposium on Cloud Computing*, p. 179–192.

[33] Huynh, L. N., Y. Lee, and R. K. Balan (2017) "DeepMon: Mobile GPU-Based Deep Learning Framework for Continuous Vision Applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, p. 82–95.

[34] Lai, Z., Y. C. Hu, Y. Cui, L. Sun, and N. Dai (2017) "Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices," in *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, p. 409–421.

[35] Liu, L., R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser (2018) "Cutting the Cord: Designing a High-Quality Untethered VR System with Low Latency Remote Rendering," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, p. 68–80.

[36] Guo, P. and W. Hu (2018) "Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications," , p. 271–284.

[37] Han, S., H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy (2016) "MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, p. 123–136.

[38] Gong, L., C. Wang, X. Li, H. Chen, and X. Zhou (2017) "A Power-Efficient and High Performance FPGA Accelerator for Convolutional Neural Networks: Work-in-Progress," in *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*.

[39] Yazdani, R., A. Segura, J. Arnau, and A. Gonzalez (2016) "An Ultra Low-power Hardware Accelerator for Automatic Speech Recognition," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 1–12.

[40] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang (2016) "A High Performance FPGA-based Accelerator for Large-scale Convolutional Neural Networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–9.

[41] Chen, T., Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam (2014) "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 269–284.

[42] Wang, J., J. Lin, and Z. Wang (2017) "Efficient Hardware Architectures for Deep Convolutional Neural Network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1941–1953.

[43] Aashish Chaubey (2020), "Downsampling and Upsampling of Images - Demystifying the Theory," `"shorturl.at/rCMPU"`.

[44] Liu, L., H. Li, and M. Gruteser (2019) "Edge Assisted Real-Time Object Detection for Mobile Augmented Reality," in *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[45] Google (2018), "Pixel Phone Hardware Tech Specs," `"https://bit.ly/397dCUB"`.

[46] Redmon, J. and A. Farhadi (2018) "YOLOv3: An Incremental Improvement," *CoRR*.

[47] Jacob Solawetz, Samrat Sahoo (2020), "Train YOLOv4-tiny on Custom Data - Lightning Fast Object Detection ," "shorturl.at/vCSVW".

[48] Liu, Y., Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang (2019) "Optimizing CNN Model Inference on CPUs," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, p. 1025–1040.

[49] Motamedi, M., D. D. Fong, and S. Ghiasi (2016) "Fast and Energy-Efficient CNN Inference on IoT Devices," *CoRR*.

[50] Wu, C., D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang (2019) "Machine Learning at Facebook: Understanding Inference at the Edge," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 331–344.

[51] Palmer, D. A. and M. Florea (2014), "Neural Processing Unit," US Patent 8,655,815.

[52] Song, Z., B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang (2020) "DRQ: Dynamic Region-Based Quantization for Deep Neural Network Acceleration," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, p. 1010–1021.

[53] Chen, Y.-H., J. Emer, and V. Sze (2016) "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *SIGARCH Comput. Archit. News*, p. 367–379.

[54] Sharma, H., J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh (2018) "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, p. 764–775.

[55] Han, S., H. Mao, and W. J. Dally (2015) "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*.

[56] Kim, Y.-D., E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin (2015) "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications," *arXiv preprint arXiv:1511.06530*.

[57] Louizos, C., K. Ullrich, and M. Welling (2017) "Bayesian Compression for Deep Learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, p. 3290–3300.

[58] Fang, B., X. Zeng, and M. Zhang (2018) "NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision," in *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, p. 115–127.

[59] Liu, S., Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du (2018) "On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, p. 389–400.

[60] Hannah Pang, Karleigh Moore, Akshay Padmanabha, Ell Rose (2021), "Feature Vector," "https://brilliant.org/wiki/feature-vector/".

[61] Yeo, H., C. J. Chong, Y. Jung, J. Ye, and D. Han (2020) "NEMO: Enabling Neural-Enhanced Video Streaming on Commodity Mobile Devices," in *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[62] Simonyan, K. and A. Zisserman (2014) "Very Deep Convolutional Networks for Large-scale Image Recognition," *arXiv preprint arXiv:1409.1556*.

[63] Klaus Hinum (2018), "Qualcomm Adreno 640," "https://www.notebookcheck.net/Qualcomm-Adreno-640-Graphics-Card.374761.0.html".

[64] Tamhankar, A. and K. R. Rao (2003) "An overview of H.264/MPEG-4 Part 10," in *Proceedings EC-VIP-MC 2003. 4th EURASIP Conference focused on Video/Image Processing and Multimedia Communications (IEEE Cat. No.03EX667)*, pp. 1–51 vol.1.

[65] Google Developers (2021), "MediaCodec," "https://developer.android.com/reference/android/media/MediaCodec".

[66] Beauchemin, S. S. and J. L. Barron (1995) "The Computation of Optical Flow," *ACM Comput. Surv.*, p. 433–466.

[67] Keymakr Inc (2020), "Image Processing Techniques: What Are Bounding Boxes?" "https://keymakr.com/blog/what-are-bounding-boxes/".

[68] MYO NeuralNet (2020), "Calculating the Output Size of Convolutions and Transpose Convolutions," "shorturl.at/ioLRV".

[69] Qualcomm Technologies Inc. (2018), "Snapdragon 845 Mobile Platform," "shorturl.at/fouyP".

[70] Liu, L. and M. T. Zsu (2009) *Encyclopedia of Database Systems*, 1st ed., Springer Publishing Company, Incorporated.

[71] Rosebrock, A. (2016), "Intersection over Union (IoU) for Object Detection," "https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/".

[72] CVLAB in EPFL (2021), "Multi-camera Pedestrians Video," "https://www.epfl.ch/labs/cvlab/data/data-pom-index-php/".

[73] Xu, Y., X. Liu, L. Qin, and S.-C. Zhu (2017) "Cross-View People Tracking by Scene-Centered Spatio-Temporal Parsing," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, p. 4299–4305.

[74] Han, B.-G., J.-G. Lee, K.-T. Lim, and D.-H. Choi (2020) "Design of a Scalable and Fast YOLO for Edge-Computing Devices," *Sensors*.

[75] Facebook's AI Research lab (2016), "PyTorch," "https://github.com/pytorch/pytorch".

[76] Bay, H., T. Tuytelaars, and L. Van Gool (2006) "Surf: Speeded up robust features," in *European conference on computer vision*, pp. 404–417.

[77] Suard, F., A. Rakotomamonjy, A. Bensrhair, and A. Broggi (2006) "Pedestrian detection using infrared images and histograms of oriented gradients," in *2006 IEEE Intelligent Vehicles Symposium*, pp. 206–212.

[78] Ying, Z., S. Zhao, S. Bhuyan, C. S. Mishra, M. T. Kandemir, and C. R. Das (2022) "Pushing Point Cloud Compression to the Edge," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 282–299.

[79] HDRsoft (2023), "What is the unit of Dynamic Range?" https://www.hdrsoft.com/resources/dri.html\#unit.

[80] Djudjic, D. (2017), "WHY DYNAMIC RANGE IS MORE IMPORTANT THAN MEGAPIXELS," https://www.diyphotography.net/dynamic-range-important-megapixels/.

[81] Blogger, B. (2023), "Top DSLR Quality Phone Cameras: An Ultimate Guide To What's Available & What's Ahead," https://www.bumblejax.com/content/top-dslr-quality-phone-cameras-2023-ultimate-guide/.

[82] Inc, A. (2023), "Adjust HDR camera settings on iPhone," https://support.apple.com/guide/iphone/adjust-hdr-camera-settings-iph2cafe2ebc/17.0/ios/17.0.

[83] Ernst, M. and B. Wronski (2021), "HDR+ with Bracketing on Pixel Phones," https://blog.research.google/2021/04/hdr-with-bracketing-on-pixel-phones.html.

[84] Guo, C. and X. Jiang (2022) "LHDR: HDR Reconstruction for Legacy Content using a Lightweight DNN," in *Proceedings of the Asian Conference on Computer Vision (ACCV)*, pp. 3155–3171.

[85] Liu, Y.-L., W.-S. Lai, Y.-S. Chen, Y.-L. Kao, M.-H. Yang, Y.-Y. Chuang, and J.-B. Huang (2020) "Single-Image HDR Reconstruction by Learning to Reverse the Camera Pipeline," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[86] Marnerides, D., T. Bashford-Rogers, J. Hatchett, and K. Debattista (2018) "Expandnet: A deep convolutional neural network for high dynamic range expansion from low dynamic range content," in *Computer Graphics Forum*, vol. 37, Wiley Online Library, pp. 37–49.

[87] Eilertsen, G., J. Kronander, G. Denes, R. K. Mantiuk, and J. Unger (2017) "HDR image reconstruction from a single exposure using deep CNNs," *ACM transactions on graphics (TOG)*, **36**(6), pp. 1–15.

[88] Corporation, N. (2023), "Jetson Orin Modules and Developer Kits," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[89] Carroll, A. and G. Heiser (2010) "An analysis of power consumption in a smartphone," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[90] Li, Z., H. Li, and L. Meng (2023) "Model Compression for Deep Neural Networks: A Survey," *Computers*, **12**(3), p. 60.

[91] Richard Durant (2021), "Pointerra: Attractive Opportunity If Growth Can Be Sustained," "https://bit.ly/3nvfRrx".

[92] Apple Inc. (2021), "https://www.apple.com/iphone-13-pro/".

[93] Marek Simonik (2021), "Record3D-Record your own 3D Videos!" "https://record3d.app/".

[94] veesus.com (2021), "INTRODUCING ZAPPCHA – MOBILE POINT CLOUD CAPTURE AND CLOUD-BASED STORAGE," "https://bit.ly/3xiSw0v".

[95] Pix4D SA (2021), "PIX4Dcatch: Turn your mobile device into a professional 3D scanner using the power of photogrammetry," "https://www.pix4d.com/product/pix4dcatch".

[96] Geo Week News Staff (2018), "Cloud Chamber: Low-cost smartphone app captures point clouds for AEC," "https://bit.ly/3DP1Qvc".

[97] Feng, Y., S. Liu, and Y. Zhu (2020) "Real-Time Spatio-Temporal LiDAR Point Cloud Compression," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10766–10773.

[98] Meagher, D. (1982) "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, pp. 129–147.

[99] de Queiroz, R. L. and P. A. Chou (2016) "Compression of 3D Point Clouds Using a Region-Adaptive Hierarchical Transform," *IEEE Transactions on Image Processing*, pp. 3947–3956.

[100] Schnabel, R. and R. Klein (2006) "Octree-Based Point-Cloud Compression," in *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, p. 111–121.

[101] Dorea, C. and R. L. de Queiroz (2018) "Block-Based Motion Estimation Speedup for Dynamic Voxelized Point Clouds," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, pp. 2964–2968.

[102] Mekuria, R., K. Blom, and P. Cesar (2017) "Design, Implementation, and Evaluation of a Point Cloud Codec for Tele-Immersive Video," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 828–842.

[103] Shao, J., H. Zhang, Y. Mao, and J. Zhang (2021) "Branchy-GNN: A Device-Edge Co-Inference Framework for Efficient Point Cloud Processing," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8488–8492.

[104] Lin, Y., Z. Zhang, H. Tang, H. Wang, and S. Han (2021) "PointAcc: Efficient Point Cloud Accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Association for Computing Machinery, New York, NY, USA, p. 449–461.

[105] Xu, T., B. Tian, and Y. Zhu (2019) "Tigris: Architecture and Algorithms for 3D Perception in Point Clouds," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, p. 629–642.

[106] He, X., H. L. Cao, and B. Zhu (2020) "Advectivenet: An eulerian-lagrangian fluidic reservoir for point cloud processing," *arXiv preprint arXiv:2002.00118*.

[107] Qi, C. R., H. Su, K. Mo, and L. J. Guibas (2017) "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[108] Point Cloud Library Contributors (2022), "Module octree - Point Cloud Library (PCL)," "https://pointclouds.org/documentation/group__octree.html".

[109] ——— (2022), "Module kdtree - Point Cloud Library (PCL)," "https://pointclouds.org/documentation/group__kdtree.html".

[110] MPEGGroup (2021), "Geometry based point cloud compression (G-PCC) test model," "https://github.com/MPEGGroup/mpeg-pcc-tmc13".

[111] Koh, N., P. K. Jayaraman, and J. Zheng (2020) "Parallel Point Cloud Compression Using Truncated Octree," in *2020 International Conference on Cyberworlds (CW)*, pp. 1–8.

[112] Jeroen Baert (2013), "Morton encoding/decoding through bit interleaving: Implementations," "https://bit.ly/3ORf506".

[113] Karras, T. (2012) "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, p. 33–37.

[114] CWI-DIS Group (2021), "cwipc-CWI Point Clouds software suite," "https://github.com/cwi-dis/cwipc".

[115] Hu, J., A. Shaikh, A. Bahremand, and R. LiKamWa (2021) "Characterizing real-time dense point cloud capture and streaming on mobile devices," in *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pp. 1–6.

[116] Charles Thomson (2019), "Reality capture 101: point clouds, photogrammetry and LiDAR," "https://bit.ly/3xfqvqy".

[117] Meynet, G., Y. Nehmé, J. Digne, and G. Lavoué (2020) "PCQM: A Full-Reference Quality Metric for Colored 3D Point Clouds," in *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*, pp. 1–6.

[118] TopoDOT Blogger (2022), "Using Point Clouds for Augmented and Virtual Reality," "https://bit.ly/3OFdA97".

[119] Mayank Raj (2020), "Point Clouds and its significance in AR," "https://bit.ly/3uknBjT".

[120] New Farmer Blogger (2015), "3D points clouds for immersive real estate and telepresence experiences," "https://bit.ly/3AhWQR5".

[121] Marek Kowalski, Jacek Naruniec (2020), "LiveScan3D-Hololens," "https://github.com/MarekKowalski/LiveScan3D-Hololens".

[122] Wang, Z.-R., C.-G. Yang, and S.-L. Dai (2020) "A Fast Compression Framework Based on 3D Point Cloud Data for Telepresence," *Int. J. Autom. Comput.*, **17**(6), p. 855–866.

[123] Mohamed, A.-M. S. (2018) "Potential of 3d laser point cloud data usage for the tourism industry," in *The International Conference on Civil and Architecture Engineering*, vol. 12, Military Technical College, pp. 1–13.

[124] Cosso, T., I. Ferrando, and A. Orlando (2015) "High-precision laser scanning for cave tourism 3D reconstruction of the Pollera cave, Italy," *GIM INTERNATIONAL-THE WORLDWIDE MAGAZINE FOR GEOMATICS*, **29**(3), pp. 23–25.

[125] Valenzuela-Urrutia, D., R. Muñoz-Riffo, and J. Ruiz-del Solar (2019) "Virtual reality-based time-delayed haptic teleoperation using point cloud data," *Journal of Intelligent & Robotic Systems*, **96**(3), pp. 387–400.

[126] Moreno, C., Y. Chen, and M. Li (2017) "A dynamic compression technique for streaming kinect-based Point Cloud data," in *2017 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, pp. 550–555.

[127] Han, B., Y. Liu, and F. Qian (2020) *ViVo: Visibility-Aware Mobile Volumetric Video Streaming*, Association for Computing Machinery.

[128] Lee, K., J. Yi, Y. Lee, S. Choi, and Y. M. Kim (2020) "GROOT: A Real-Time Streaming System of High-Fidelity Volumetric Videos," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, Association for Computing Machinery, New York, NY, USA.

[129] We are Luminous (2016), "Interitum - Sureal VR Point Cloud Game," `"https://www.youtube.com/watch?v=P5BgrdXis68"`.

[130] Ayukawa, S., N. Tokudome, S. Enokida, and T. Nishida (2016) "Time-series LIDAR data superimposition for autonomous driving," *Proc. of ICT-ROBOT, ThBT3*, **3**.

[131] Aksoy, E. E., S. Baci, and S. Cavdar (2020) "Salsanet: Fast road and vehicle segmentation in lidar point clouds for autonomous driving," in *2020 IEEE intelligent vehicles symposium (IV)*, IEEE, pp. 926–932.

[132] Teixeira, M. A. S., H. B. Santos, A. S. d. Oliveira, L. V. Arruda, and F. Neves (2017) "Robots perception through 3d point cloud sensors," in *Robot Operating System (ROS)*, Springer, pp. 525–561.

[133] Kuntz, A., C. Bowen, and R. Alterovitz (2020) "Fast anytime motion planning in point clouds by interleaving sampling and interior point optimization," in *Robotics Research*, Springer, pp. 929–945.

[134] Liu, M. (2015) "Robotic online path planning on point cloud," *IEEE transactions on cybernetics*, **46**(5), pp. 1217–1228.

[135] Park, Jounsup and Chou, Philip A. and Hwang, Jenq-Neng (2018), "Rate-Utility Optimized Streaming of Volumetric Media for Augmented Reality," .

[136] Qian, F., B. Han, J. Pair, and V. Gopalakrishnan (2019) "Toward Practical Volumetric Video Streaming on Commodity Smartphones," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, p. 135–140.

[137] Qi, C. R., L. Yi, H. Su, and L. J. Guibas (2017) "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, p. 5105–5114.

[138] Yan, X., C. Zheng, Z. Li, S. Wang, and S. Cui (2020) "PointASNL: Robust Point Clouds Processing Using Nonlocal Neural Networks With Adaptive Sampling," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 5588–5597.

[139] Wu, W., Z. Qi, and L. Fuxin (2019) "PointConv: Deep Convolutional Networks on 3D Point Clouds," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 9613–9622.

[140] Qi, C. R., W. Liu, C. Wu, H. Su, and L. J. Guibas (2018) "Frustum PointNets for 3D Object Detection from RGB-D Data," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 918–927.

[141] Li, B. (2017) "3D Fully Convolutional Network for Vehicle Detection in Point Cloud," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE Press, p. 1513–1518.

[142] Simon, M., K. Amende, A. Kraus, J. Honer, T. Samann, H. Kaulbersch, S. Milz, and H. Michael Gross (2019) "Complexer-YOLO: Real-Time 3D Object Detection and Tracking on Semantic Point Clouds," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1190–1199.

[143] Giancola, S., J. Zarzar, and B. Ghanem (2019) "Leveraging Shape Completion for 3D Siamese Tracking," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1359–1368.

[144] Sun, J., Y. Xie, S. Zhang, G. Zhang, H. Bao, and X. Zhou (2021) "You Don't Only Look Once: Constructing Spatial-Temporal Memory for Integrated 3D

Object Detection and Tracking," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 3265–3174.

[145] Choy, C., J. Gwak, and S. Savarese (2019) "4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 3070–3079.

[146] Feng, Y., B. Tian, T. Xu, P. Whatmough, and Y. Zhu (2020) "Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1037–1050.

[147] Elseberg, J., D. Borrmann, and A. Nüchter (2013) "One billion points in the cloud – an octree for efficient processing of 3D laser scans," *ISPRS Journal of Photogrammetry and Remote Sensing*, pp. 76–88.

[148] Google (2021), "Draco," `"https://github.com/google/draco"`.

[149] MPEG (2019), "G-PCC codec description v2," `"https://bit.ly/3nN43C8"`.

[150] TheAILearner Blogger (2018), "Image Processing – Nearest Neighbour Interpolation," `"https://bit.ly/3l4iY95"`.

[151] de Queiroz, R. L. and P. A. Chou (2017) "Motion-Compensated Compression of Dynamic Voxelized Point Clouds," *IEEE Transactions on Image Processing*, **26**(8), pp. 3886–3895.

[152] Santos, C., M. Gonçalves, G. Corrêa, and M. Porto (2021) "Block-Based Inter-Frame Prediction For Dynamic Point Cloud Compression," in *2021 IEEE International Conference on Image Processing (ICIP)*, pp. 3388–3392.

[153] Wu, C.-H., C.-F. Hsu, T.-K. Hung, C. Griwodz, W. T. Ooi, and C.-H. Hsu (2022) "Quantitative Comparison of Point Cloud Compression Algorithms with PCC Arena," *IEEE Transactions on Multimedia*, pp. 1–1.

[154] Liu, H., H. Yuan, Q. Liu, J. Hou, and J. Liu (2020) "A Comprehensive Study and Comparison of Core Technologies for MPEG 3-D Point Cloud Compression," *IEEE Transactions on Broadcasting*, pp. 701–717.

[155] Schwarz, S., M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. A. Chou, R. A. Cohen, M. Krivokuća, S. Lasserre, Z. Li, J. Llach, K. Mammou, R. Mekuria, O. Nakagami, E. Siahaan, A. Tabatabai, A. M. Tourapis, and V. Zakharchenko (2019) "Emerging MPEG Standards for Point Cloud Compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pp. 133–148.

[156] Jang, E. S., 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*, pp. 118–123.

[157] Kim, J., J. Im, S. Rhyu, and K. Kim (2020) "3D Motion Estimation and Compensation Method for Video-Based Point Cloud Compression," *IEEE Access*, pp. 83538–83547.

[158] Li, L., Z. Li, V. Zakharchenko, J. Chen, and H. Li (2020) "Advanced 3D Motion Prediction for Video-Based Dynamic Point Cloud Compression," *IEEE Transactions on Image Processing*, pp. 289–302.

[159] MPEG (2022), "MPEG Point Cloud Compression," `"https://mpeg-pcc.org"`.

[160] Huang, T. and Y. Liu (2019) "3D Point Cloud Geometry Compression on Deep Learning," in *Proceedings of the 27th ACM International Conference on Multimedia*, p. 890–898.

[161] Tu, C., E. Takeuchi, A. Carballo, and K. Takeda (2019) "Point Cloud Compression for 3D LiDAR Sensor using Recurrent Neural Network with Residual Blocks," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 3274–3280.

[162] Que, Z., G. Lu, and D. Xu (2021) "VoxelContext-Net: An Octree Based Framework for Point Cloud Compression," in *CVPR*, pp. 6042–6051.

[163] Huang, L., S. Wang, K. Wong, J. Liu, and R. Urtasun (2020) "OctSqueeze: Octree-Structured Entropy Model for LiDAR Compression," in *CVPR*.

[164] Rusu, R. B. and S. Cousins (2011) "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*.

[165] Bédorf, J., E. Gaburov, and S. P. Zwart (2012) "A sparse octree gravitational N-body code that runs entirely on the GPU processor," *Journal of Computational Physics*, pp. 2825–2839.

[166] Le Gall, D. (1991) "MPEG: A Video Compression Standard for Multimedia Applications," *Commun. ACM*, p. 46–58.

[167] Pearlman, W. A. and A. Said (2011) *Entropy coding techniques*, Cambridge University Press, p. 41–76.

[168] Langdon, G. G. (1984) "An Introduction to Arithmetic Coding," *IBM Journal of Research and Development*, pp. 135–149.

[169] Point Cloud Library Contributors (2022), "PCL GPU Octree," `"https://github.com/PointCloudLibrary/pcl/tree/master/gpu/octree"`.

[170] Shumaker, R. and L. Stephanie (2014) *Virtual, Augmented and Mixed Reality: Designing and Developing Augmented and Virtual Environments: 6th International Conference, VAMR 2014, Held as Part of HCI International 2014, Heraklion, Crete, Greece, June 22-27, 2014, Proceedings, Part I*, Springer.

[171] Beg, M., Y. C. Chang, and T. F. Tang (2002) "Performance evaluation of error resilient tools for MPEG-4 video transmission over a mobile channel," in *2002 IEEE International Conference on Personal Wireless Communications*, pp. 285–289.

[172] Vision Lab, Nanjing University (2020), "Multiscale Point Cloud Geometry Compression," "https://bit.ly/3xiAxah".

[173] Besl, P. J. and N. D. McKay (1992) "Method for registration of 3-D shapes," in *Sensor fusion IV: control paradigms and data structures*, pp. 586–606.

[174] Corning, A. (2021), "AR/VR in the OR – Surgical Applications of Augmented and Virtual Reality," "shorturl.at/fqwAJ".

[175] Andrei Frumusanu (2021), "The Snapdragon 888 vs The Exynos 2100: Cortex-X1 & 5nm, Who Does It Better?" "https://bit.ly/3OF66Tw".

[176] Ricci Rox (2022), "Power-hungry Snapdragon 8 Gen 1 gets trounced by the Apple A15 Bionic in real-world gaming test," "https://bit.ly/3P086pp".

[177] Zhao, S., H. Zhang, S. Bhuyan, C. S. Mishra, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das (2020) "DéJà View: Spatio-Temporal Compute Reuse for Energy-Efficient 360° VR Video Streaming," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, IEEE Press, p. 241–253.

[178] Zhao, S., H. Zhang, C. S. Mishra, S. Bhuyan, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. Das (2021) "HoloAR: On-the-Fly Optimization of 3D Holographic Processing for Augmented Reality," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Association for Computing Machinery, New York, NY, USA, p. 494–506.

[179] Romphf, J., E. Neuman-Donihue, G. Heyworth, and Y. Zhu (2021) "Resurrect3D: An Open and Customizable Platform for Visualizing and Analyzing Cultural Heritage Artifacts," in *The 26th International Conference on 3D Web Technology*, Association for Computing Machinery, New York, NY, USA.

[180] TruePoint Laser Scanning, LLC (2022), "What Is 3D Laser Scanning?" "https://www.truepointscanning.com/what-is-3d-laser-scanning".

[181] Alan Walford (2017), "What is Photogrammetry?" "https://www.photogrammetry.com/".

[182] Microsoft (2022), "Kinect for Windows," "https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows".

[183] Intel Corporation (2022), "Intel RealSense Depth and Tracking cameras ," "https://www.intelrealsense.com/".

[184] Apple Inc. (2022), "iPhone 13 Pro," "https://www.apple.com/am/iphone-13-pro/".

[185] SAMSUNG (2022), "Specifications | Galaxy S20, S20+ and S20 Ultra - Samsung," "https://www.samsung.com/levant/smartphones/galaxy-s20/specs/".

[186] GlobeNewswire, Inc. (2022), "Straits Research," "https://www.globenewswire.com/en/news-release/2022/09/14/2516327/0/en/LIDAR-Market-Size-is-projected-to-reach-USD-6-93-Billion-by-2030-growing-at-a html".

[187] Liu, Z., Q. Li, X. Chen, C. Wu, S. Ishihara, J. Li, and Y. Ji (2021) "Point cloud video streaming: Challenges and solutions," *IEEE Network*, **35**(5), pp. 202–209.

[188] Mekuria, R., K. Blom, and P. Cesar (2016) "Design, implementation, and evaluation of a point cloud codec for tele-immersive video," *IEEE Transactions on Circuits and Systems for Video Technology*, **27**(4), pp. 828–842.

[189] Bhuyan, S., S. Zhao, Z. Ying, M. T. Kandemir, and C. R. Das (2022) "End-to-End Characterization of Game Streaming Applications on Mobile Platforms," *Proc. ACM Meas. Anal. Comput. Syst.*, **6**(1).

[190] Lai, Z., Y. C. Hu, Y. Cui, L. Sun, and N. Dai (2017) "Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Association for Computing Machinery, New York, NY, USA, p. 409–421.

[191] Ying, Z., S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das (2022) "Exploiting Frame Similarity for Efficient Inference on Edge Devices," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 1073–1084.

[192] Sarma, A., S. Singh, H. Jiang, A. Pattnaik, A. K. Mishra, V. Narayanan, M. T. Kandemir, and C. R. Das (2021) "Exploiting Activation based Gradient Output Sparsity to Accelerate Backpropagation in CNNs," *CoRR*, **abs/2109.07710**.

[193] Singh, S., A. Sarma, S. Lu, A. Sengupta, M. T. Kandemir, E. Neftci, V. Narayanan, and C. R. Das (2022) "Skipper: Enabling efficient SNN training through activation-checkpointing and time-skipping," in *2022 55th IEEE/ACM*

International Symposium on Microarchitecture (MICRO), IEEE Computer Society, Los Alamitos, CA, USA, pp. 565–581.

[194] Imani, M., M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing (2018) "RAPIDNN: In-Memory Deep Neural Network Acceleration Framework," *CoRR*, **abs/1806.05794**.

[195] Komarichev, A., Z. Zhong, and J. Hua (2019) "A-CNN: Annularly Convolutional Neural Networks on Point Clouds," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 7413–7422.

[196] Liu, X., C. R. Qi, and L. J. Guibas (2019) "FlowNet3D: Learning Scene Flow in 3D Point Clouds," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 529–537.

[197] Hashimoto, T. and M. Saito (2019) "Normal Estimation for Accurate 3D Mesh Reconstruction with Point Cloud Model Incorporating Spatial Structure," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2019, Long Beach, CA, USA, June 16-20, 2019*, Computer Vision Foundation / IEEE, pp. 54–63.

[198] Wang, X., Y. Xu, K. Xu, A. Tagliasacchi, B. Zhou, A. Mahdavi-Amiri, and H. Zhang (2020) "Pie-net: Parametric inference of point cloud edges," *Advances in neural information processing systems*, **33**, pp. 20167–20178.

[199] Jeroen Baert (2013), "Morton encoding/decoding through bit interleaving: Implementations," `"https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/"`.

[200] Zhang, J.-F. and Z. 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*, Association for Computing Machinery, New York, NY, USA, p. 1078–1090.

[201] Feng, Y., G. Hammonds, Y. Gan, and Y. Zhu (2022) "Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, NY, USA, p. 962–977.

[202] NVIDIA Corporation (2022), "Jetson AGX Xavier Series," `"https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/"`.

[203] Lan, S., R. Yu, G. Yu, and L. S. Davis (2019) "Modeling Local Geometric Structure of 3D Point Clouds Using Geo-CNN," in *2019 IEEE/CVF Conference*

on Computer Vision and Pattern Recognition (CVPR), IEEE Computer Society, Los Alamitos, CA, USA, pp. 998–1008.

[204] Blanc, T., M. El Beheiry, C. Caporal, J.-B. Masson, and B. Hajj (2020) "Genuage: visualize and analyze multidimensional single-molecule point cloud data in virtual reality," *Nature Methods*, **17**(11), pp. 1100–1102.

[205] Raj, M. (2020), "Point Clouds and it's significance in AR!" "https://medium.com/arway/point-clouds-and-its-significance-in-ar-155db2673865".

[206] VREX (2022), "How to Bring Point Clouds into VR," "https://www.vrex.no/blog/point-cloud-vr/".

[207] Chen, J., Z. Kira, and Y. K. Cho (2019) "Deep learning approach to point cloud scene understanding for automated scan to 3D reconstruction," *Journal of Computing in Civil Engineering*, **33**(4), p. 04019027.

[208] Lin, C.-H., C. Kong, and S. Lucey (2018) "Learning Efficient Point Cloud Generation for Dense 3D Object Reconstruction," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI Press.

[209] Liu, X., W. Zheng, Y. Mou, Y. Li, and L. Yin (2021) "Microscopic 3D reconstruction based on point cloud data generated using defocused images," *Measurement and Control*, **54**(9-10), pp. 1309–1318.

[210] Geiger, A., P. Lenz, and R. Urtasun (2012) "Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, USA, p. 3354–3361.

[211] ByteBridge (2021), "How 3D Point Cloud Annotation Service Fuels the Field of Automatic Driving?" "https://medium.com/nerd-for-tech/application-of-3d-point-cloud-in-the-field-of-automatic-driving-723ec9544a6c".

[212] Chen, S., B. Liu, C. Feng, C. Vallespi-Gonzalez, and C. Wellington (2020) "3D Point Cloud Processing and Learning for Autonomous Driving: Impacting Map Creation, Localization, and Perception," *IEEE Audio and Electroacoustics Newsletter*, **38**(1), pp. 68–86.

[213] Chen, X., H. Ma, J. Wan, B. Li, and T. Xia (2017) "Multi-view 3D Object Detection Network for Autonomous Driving," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 6526–6534.

[214] Qi, C. R., H. Su, K. Mo, and L. J. Guibas (2017) "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 77–85.

[215] You, H., Y. Feng, R. Ji, and Y. Gao (2018) "PVNet: A Joint Convolutional Network of Point Cloud and Multi-View for 3D Shape Recognition," in *Proceedings of the 26th ACM International Conference on Multimedia*, Association for Computing Machinery, New York, NY, USA, p. 1310–1318.

[216] Zhou, Y. and O. Tuzel (2018) "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 4490–4499.

[217] Schulman, J., A. Lee, J. Ho, and P. Abbeel (2013) "Tracking deformable objects with point clouds," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, pp. 1130–1137.

[218] Landrieu, L. and M. Simonovsky (2018) "Large-Scale Point Cloud Semantic Segmentation with Superpoint Graphs," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 4558–4567.

[219] Eldar, Y., M. Lindenbaum, M. Porat, and Y. Y. Zeevi (1997) "The farthest point strategy for progressive image sampling," *IEEE Transactions on Image Processing*, **6**(9), pp. 1305–1315.

[220] Omohundro, S. M. (1989) *Five balltree construction algorithms*, International Computer Science Institute Berkeley.

[221] Duda, R. O., P. E. Hart, and D. G. Stork (1973) *Pattern classification and scene analysis*, Wiley New York.

[222] Wu, Z., S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao (2022), "Princeton Model Net," `"https://modelnet.cs.princeton.edu/"`.

[223] Dai, A., A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Niessner (2017) "Scannet: Richly-annotated 3d reconstructions of indoor scenes," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 2432–2443.

[224] Zhu, Y. (2022) "RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Association for Computing Machinery, New York, NY, USA, p. 76–89.

[225] Jan Bender, Weiler Marcel and Stephan Seitz (2022), "cuNSearch," https://github.com/InteractiveComputerGraphics/cuNSearch.

[226] Rama C. Hoetzlein (2014), "Fast Fixed-Radius Nearest Neighbor Search on the GPU," https://tinyurl.com/4rcjdu7p.

[227] Gross, J., M. Köster, and A. Krüger (2019) "Fast and Efficient Nearest Neighbor Search for Particle Simulations." in *CGVC*, pp. 55–63.

[228] Lixin Xue and Oliver Batchelor (2022), "FRNN," https://github.com/lxxue/FRNN.

[229] Connor, M. F. (2007) *Simple, Thread-Safe, Approximate Nearest Neighbor Algorithm*, Master's thesis.

[230] DISCOVER three.js (2022), "A Brief Introduction to Texture Mapping," "https://discoverthreejs.com/book/first-steps/textures-intro/".

[231] Rubinsztejn, A. (2018), "What Is The N-body Problem?" "https://gereshes.com/2018/05/07/what-is-the-n-body-problem/".

[232] Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson (2009) "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, Association for Computing Machinery, New York, NY, USA, p. 233–244.

[233] NVIDIA Corporation (2022), "NVIDIA Tensor Cores," "https://www.nvidia.com/en-us/data-center/tensor-cores/".

[234] Armeni, I., O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese (2016) "3D Semantic Parsing of Large-Scale Indoor Spaces," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1534–1543.

[235] Wu, Z., S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao (2015) "3D ShapeNets: A deep representation for volumetric shapes," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1912–1920.

[236] NVIDIA Corporation (2019), "tegrastats Utility," "https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html##page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html".

Vita

Ziyu Ying

The details of my childhood are inconsequential.