A Scalable and Programmable I/O Controller for Region-based Computing

by

Van Nguyen

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2020 by the
Graduate Supervisory Committee:

Robert LiKamWa, Chair
Suren Jayasuriya
Yezhou Yang

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

I present my work on a scalable and programmable I/O controller for region-based computing, which will be used in a rhythmic pixel-based camera pipeline. I provide a breakdown of the development and design of the I/O controller and how it fits in to rhythmic pixel regions, along with a study on memory traffic of rhythmic pixel regions and how this translates to energy efficiency. This rhythmic pixel region-based camera pipeline has been jointly developed through Dr. Robert LiKamWa's research lab. High spatiotemporal resolutions allow high precision for vision applications, such as for detecting features for augmented reality or face detection. High spatiotemporal resolution also comes with high memory throughput, leading to higher energy usage. This creates a tradeoff between high precision and energy efficiency, which becomes more important in mobile systems. In addition, not all pixels in a frame are necessary for the vision application, such as pixels that make up the background. Rhythmic pixel regions aim to reduce the tradeoff by creating a pipeline that allows an application developer to specify regions to capture at a non-uniform spatiotemporal resolution. This is accomplished by encoding the incoming image, and only sending the pixels within these specified regions. Later these encoded representations will be decoded to a standard frame representation usable by traditional vision applications. My contribution to this effort has been the design, testing and evaluation of the I/O controller.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Modern image sensors are outputting at higher resolutions and higher framerates to allow for clearer images and higher accuracy on visual tasks. This in turn generates higher memory traffic, resulting in higher energy consumption. This can be problematic for systems that are attempting to be energy efficient, or systems that have lower computational overhead. Mobile systems are a prime example of a limited system due to their reduced computation abilities, small battery sizes and heat management requirements. In addition, increasingly complex tasks are favored on mobile systems, such as augmented reality or real-time face detection. Because of these conflicting factors, mobile systems are forced to spatiotemporally downscale image and video capture to save energy and reduce system load (Jinhan Hu, 2018).

Reducing resolution also reduces memory traffic over DRAM-based frame buffers, providing the previously stated tradeoff of energy efficiency for task precision. Unfortunately, reducing resolution in a typical image pipeline must be done over the entire frame, and thus reduces task precision across the entire frame as well. Here we introduce rhythmic pixel regions, a novel image pipeline that takes advantage of spatial temporal redundancies to reduce memory overhead. Rhythmic pixel regions allow application developers to specify a set of regions each with their own unique spatiotemporal resolution to be transferred by the pipeline. Thus, pixels that are not necessary for the task at hand can be discarded to reduce memory throughput. With rhythmic pixel regions, this tradeoff of energy efficiency for task precision can instead be

done on regions of the frame where detail is not as important, and thus task precision across the whole frame will not be impacted as negatively. For example, for face detection, the background of an image is not important for the task and does not need to be sent over the memory interface, reducing memory traffic and thus power consumption.

The key insight to be drawn from these observations is that the current paradigm of processing frames at a uniform resolution and uniform frame rate is limiting the performance and efficiency of visual systems. More fine-grained control will allow better efficiency at minimal performance cost.

The core of rhythmic pixel regions are two I/O interfaces, one for encoding the input image into a compact stream for storage in DRAM, and one for decoding this compact representation into a full frame able to be utilized by standard visual computing algorithms. These two in combination allow compact representations of the frame to be written to memory, while still providing full frames when needed. Since the decoding is done as requested and sent to the application in a streaming fashion, there is no additional memory overhead for this full frame representation.

Rhythmic pixel regions differ from region-of-interest (ROI) based computing by supporting a high number of regions, where each region has its own individually configurable size, position, and spatiotemporal resolution. These design choices allow fine-grained control of what pixels are sent over the DDR interface, allowing high spatiotemporal resolution capture where it matters with reduced power consumption than a traditional pipeline. Thus, the focus of the design is scalability and programmability.

Both I/O controllers are currently being implemented on a Xilinx ZCU102 FPGA SoC platform. In addition, memory traffic simulations were also performed on various

visual tasks with a simulated rhythmic pixel region pipeline to estimate memory traffic and energy savings using a DRAM simulator (CMU-SAFARI, n.d.) combined with a DRAM power simulator (tukl-msd, n.d.).

In this thesis, I will go into further detail discussing the design choices and evolution of the first I/O selection interface used to encode the frame. In addition, I will discuss and analyze the results of memory traffic simulation on rhythmic pixel regions and its impact on energy utilization.

CHAPTER 2

BACKGROUND/RELATED WORK

Image pipelines collect digital readings of pixel data in a frame from a sensor and typically passes it through an image signal processor (ISP) to do some visual improvements, such as white balance or format changes. After these modifications, the full frame is written to DRAM and will signal that a frame is available in memory. Certain visual tasks, such as augmented reality, can greatly benefit from high spatiotemporal resolutions. However, these high fidelity video streams incur high data movement across the DDR interface, which significantly increases energy consumption (Saugata Ghose, 2018).

Rhythmic pixel regions can be compared top multi-ROI sensors as both are able to select important regions of the image to be stored, rather than storing the full frame. In multi-ROI sensors, selection happens at the sensor level, which introduces significant compromises due to circuit complexity. For example, Ximea multi-ROI cameras only support 4 non-overlapping regions, with no customization available for region framerate or resolution (Ximea, n.d.). In comparison, rhythmic pixel regions select pixels immediately after the ISP, allowing traditional image sensors to be used, and thus eliminating the limitations imposed by the complex circuitry of multi-ROI sensors. This design choice allows rhythmic pixel regions to offer 200 regions that can overlap and be placed anywhere, all with individual sizes, framerate, and resolution settings.

Rhythmic pixel regions can also be compared to video compression algorithms such as H.265 (Wikipedia, n.d.) which utilize estimated motion between frames to reduce redundant information transfer. These implementations rely on having copies of the

current, and sometimes previous, frame in memory. This still means the full frame is transferred over the DRAM interface whereas rhythmic pixel regions use less memory overhead by encoding the frame before it passes over the interface. In addition, some of the techniques applied by these codecs can be complex and expensive, compared to rhythmic pixel regions simply discarding unnecessary pixels.

CHAPTER 3

DESIGN

The first I/O selection interface, also known as the encoder, is currently implemented in Xilinx Vivado HLS. Vivado HLS takes code written in C++ and converts it to an RTL implementation usable on an FPGA board. The code is written like standard C++ with some modifications using Vivado HLS pragmas and other supported libraries. These Vivado HLS pragmas allow efficient parallel and pipelined execution of code. For example, the encoder makes heavy use of the PIPELINE pragma, which allows multiple iterations of a loop to occur concurrently rather than one after another. This pragma allows our design to accept new pixel data every clock cycle, which is a requirement of the ISP. In addition, the encoder utilizes the UNROLL pragma to allow iterations of a loop to occur in full parallel. This is utilized to allow us to check all regions in parallel, heavily speeding up the design.

Vivado HLS also allows testing of the encoder by analyzing the waveforms of the AXI streams generated by the encoder. AXI streams allow the encoder to transfer data between blocks without going over the DRAM interface. In this case, the encoder outputs three AXI streams to three different DMA blocks so that the contents of the stream can be written to memory. The first stream is the pixel stream, used to carry the encoded pixel data. The second and third stream contain the row offset data and EncMask data, respectively. These AXI streams contain various signals to ensure valid transfer, and these can be analyzed to ensure that the IP block is outputting correctly. Notably important for the encoder are the TVALID, TREADY and TLAST signals. TVALID indicates that the master block is outputting a valid signal, so if the encoder is not

asserting TVALID, the output signal is not correct. TREADY indicates that the encoder is ready to accept a new transfer of data. Specifically, this means the encoder is ready to read in new pixel data. If TREADY is not asserted on every cycle, then the block will not meet the timing requirements for the ISP and the design will not run on hardware. Finally, the TLAST signal on a pixel stream indicates that this is the last pixel in the row. Because the encoder does not send every pixel, it must modify when this TLAST signal is asserted on the output. Thus, there is a counter within the code that tracks when the last pixel of the encoded row is sent and asserts TLAST appropriately based on this counter.

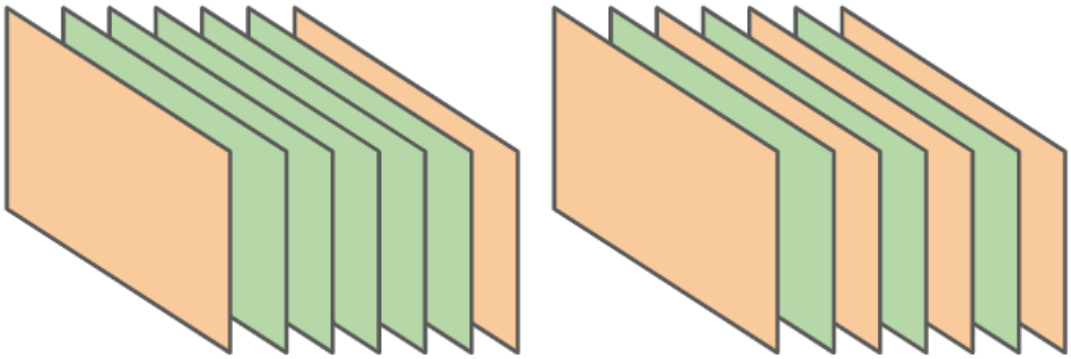
After thorough functional testing and waveform analysis, the encoder is exported as an RTL design to be implemented in Xilinx Vivado. Our Vivado design is based off the ReVISION stack (Xilinx, Embedded Reference Platforms User Guide, n.d.) provided by Xilinx. The ReVISION stack provides a fully functioning hardware design for the ZCU102 with DPU support for neural network-based algorithms. In addition, the ReVISION stack has working support for a MIPI CSI-2 camera interface and HDMI out, allowing rapid prototyping and testing of the encoder. We currently insert the encoder just after the traditional image signal pipeline processes, right before the frame is entered into memory. This allows the ISP processes to work properly, as these algorithms, such as demosaic, rely on pixel neighbors, which can be disturbed by the encoding process. Additionally, the memory overhead of the ISP is minimal, as the frame is not written into memory until after these operations. Inserting after the ISP also allows seamless integration of rhythmic pixel regions with existing ISPs without any modification required for the algorithms. Once the encoder design has been integrated, the hardware design can be exported and run on the Xilinx Zynq UltraSCALE+ MPSoC ZCU102

FPGA board (Xilinx, Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, n.d.) for hardware testing. In addition, at this stage, we can evaluate the hardware design for resource utilization and estimated power consumption.

The encoder uses region information determined by the vision application to decimate the pixel stream to a decreased size. The encoder works by processing the pixels of a frame in raster scan format, as the ISP provides the pixel data in an AXI4-Stream, which is a FIFO buffer. Rhythmic pixel regions also utilize a cycle length factor, which tells the pipeline how often to capture a full frame with no removed pixels to allow for application developers to check the entire frame for important pixels. For example, at a cycle length of 5, the encoder will send a full frame capture every 5 frames, and for the other frames it will decimate the pixel stream according to the region descriptors.

Figure 1 illustrates the concept, where the orange frames indicate the full frame captures, and the green frames indicate those captured according to the provided region data. For a scene with minimal change, such as security camera footage of a hallway, a cycle length of 5 as shown on the left will be adequate. Since there is a low chance of new points of interest appearing in the hallway, the encoder does not need to capture all the details as often. However, for a rapidly changing scene, such as a busy sidewalk in a city, it might be more appropriate to have a cycle length of 1 as shown on the right, so that new features can be detected as soon as possible to ensure high task accuracy.

Figure 1: Cycle Length Demonstration



The first iteration of the encoder sent all pixels across the interface but zeroed out the data of pixels that were not selected in a region. The idea was that repeated zeros sent over the memory interface would save power, but that was not the case. The power utilization of the design remained the same no matter how many pixels were zeroed out.

The next implementation of the encoder only sent pixels that were selected by the controller, which significantly reduced memory load as intended. Various optimizations were made here in HLS to speed up the execution of the encoder. In addition, at this stage, the method of packing the encoded stream was decided. Two options were considered, one being sending only the pixels requested in a raster scan fashion, which creates an encoded format that cannot be understood by traditional vision applications. The second option was packing the pixels in groups as assigned by their regions, so that the encoded format would still be able to be processed by vision applications. This option proved non-ideal for two reasons. First, the algorithm to pack the regions in a manner that made sense would require iterating over the image multiple times or iterating over the image in a non-raster scan fashion, which is incompatible with the pixel stream input to

the encoder. Second, this packing method would consume more space in memory, as there would need to be additional padded pixels to ensure the regions were logically placed.

Finally, to improve scalability of the encoder the regions are passed in grouped as “chunks”. For example, regions covering any of the first quarter of the image’s rows are considered in chunk 0. The regions covering any of the second quarter of the rows are considered in chunk 1, and so on until chunk 3. Regions may be placed into multiple chunks if the region covers multiple chunks. This chunking allows us to speed up the check of whether the pixel is in a region by reducing the number of regions to be checked while allowing us to support more overall regions across the entire frame with minimal latency increase.

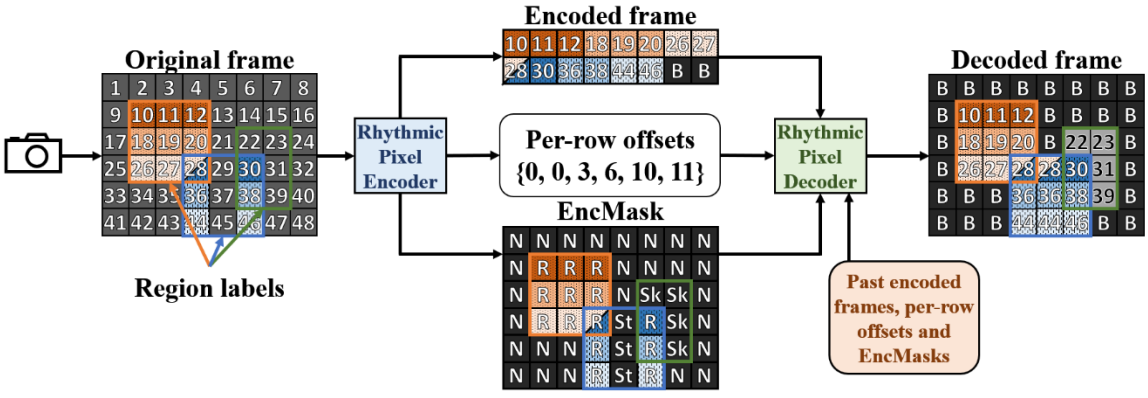
The decoder must be able to service pixel requests within a random portion of the frame rather than raster scan order, due to the memory access patterns of the vision application and hardware. To do this, the encoder generates some metadata about the encoded frame to be utilized later by the decoder. As the encoder is determining whether a pixel is within a region, it also generates a per-row offset used to indicate the number of encoded pixels prior to that row along with an EncMask. The EncMask is a 2-bit array of flags that indicate how a pixel in the original frame is sampled by the encoder in space and time:

- N (00): Non-regional pixel
- St (01): Regional pixel but strided
- Sk (10): Regional pixel but temporally skipped
- R (11): Regional pixel

Pixels that are strided are generated in the decoded image by copying the nearest encoded pixel. Pixels that are temporally skipped are generated in the decoded image by copying a pixel in the same position from a previous frame. Thus, a cache of previous encoded frames, per-row offsets and EncMasks must be kept for the decoding process. Using this metadata rather than the region information allows the decoder to service pixel requests anywhere in the image much faster than utilizing a raster scan method like the encoder.

Figure 2 and Figure 3 demonstrate the rhythmic pixel region flow. The original frame is encoded by only sending the pixels that fall within the specified regions and their parameters. The rhythmic pixel encoder also outputs a set of per-row offsets and an EncMask containing information about the encoded status of the original pixels. These items are sent to the decoder and are used alongside a cache of previous encoded frames, per-row offsets and EncMasks to generate a decoded frame.

Figure 2: Demonstration Frame Encoding/Decoding



Specifically, in Figure 2 we can see three different regions. The orange region is a full resolution region, so all pixels are selected. The blue region has a stride of 1, so the middle column of the region is not encoded. Finally, the green region was temporally

skipped on this frame, so no pixels are encoded. The encoder also generates the per-row offsets and EncMask as shown in the middle of the figure. The abbreviations from above are shown on the EncMask for clarity. These are fed into the decoder, along with the previous encoded frames, per-row offsets and EncMasks. The previous metadata and frame data are used to fill in pixels that are temporally skipped, such as in this example. Pixels 22, 23, 31 and 39 are filled in from a previous cached frame since the original pixels were temporally skipped. Pixels 28, 36 and 44 were filled in by copying from the neighboring pixel to the left, since the original pixels were strided away in the encoded frame.

Figure 3: Real Frame Encoding/Decoding

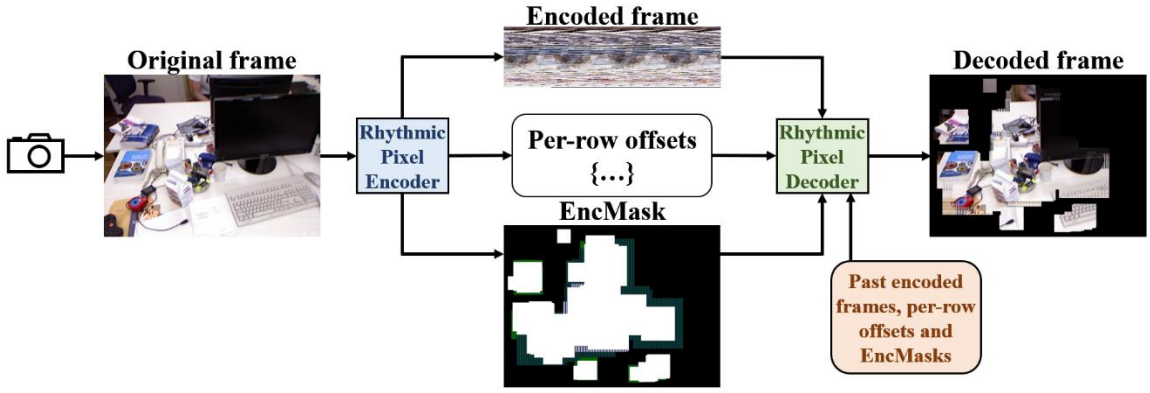


Figure 3 shows the same process as Figure 2, except on a real image that was processed using ORB SLAM and then passed through the rhythmic pixel region system. This example clearly shows how the encoded frame is a distorted representation and cannot be used in traditional vision applications. However, the decoded frame looks simply like a masked version of the original frame, allowing vision applications to work

as normal. In this example, the EncMask is depicted using colored pixels, where white is a regional pixel, blue is a strided pixel, green is a skipped pixel and black is a non-regional pixel.

CHAPTER 4

RESULTS

In order to simulate memory traffic, I first had to create a simulator for rhythmic pixel regions that allows the user to configure a set of frames to be loaded as a video stream, along with allowing a set of configuration files to be loaded as the ROIs to be used by the I/O selection interface. This simulator is implemented in C++, using the OpenCV library to perform image modifications. The simulator first reads in a frame, along with a corresponding CSV file that will describe the regions of the frame. In the case of a blank CSV file, a full frame is captured. Then, the simulator will simulator the encoder by iterating over all pixels and determining the EncMask and row offsets as it iterates, while also recording any memory operations that are performed, such as writing encoded pixels, writing the EncMask or writing the row offset value. These memory operations are written as memory traces compatible with Ramulator, therefore only an address and operation (read or write) is specified. Ramulator and DRAMPower are configured with 8-bit transaction width, and so for every 8-bit transaction in the simulator, there is a memory trace generated. In addition, the simulator will simulate decoder DRAM read transactions. These transactions include reading the EncMasks of the row, reading the row offset of the row and reading the actual pixel data. Because the encoder and decoder rely on a 2 pixel per clock timing restriction, the simulator also does transfers 2 pixels at a time.

The number of read and write operations are counted in the simulator, including the operations incurred by writing the metadata. When passing this read/write count data into the DRAM simulator and DRAM power simulator, we noticed that power directly

corresponded to read and write operations, matching our previous assumptions. In the below figures, I provide estimated memory throughput for 3 workloads and their respective datasets.

Table 1: Workload Descriptions

Task	Algorithm	Resolution	Benchmark	# Frames
Visual SLAM	ORB-SLAM (Raul Mur-Artal, 2015)	4K @ 30 fps	In-house dataset	6000
Pose estimation	PoseNet (Zhe Cao, 2018)	720p @ 30 fps	PoseTrack 2017 (Andriluka, 2018)	3792
Face detection	RetinaNet (1996scarlet, n.d.)	SVGA @ 30 fps	ChokePoint dataset (Y. Wong, 2011)	22099

Figure 4: Visual SLAM Throughput

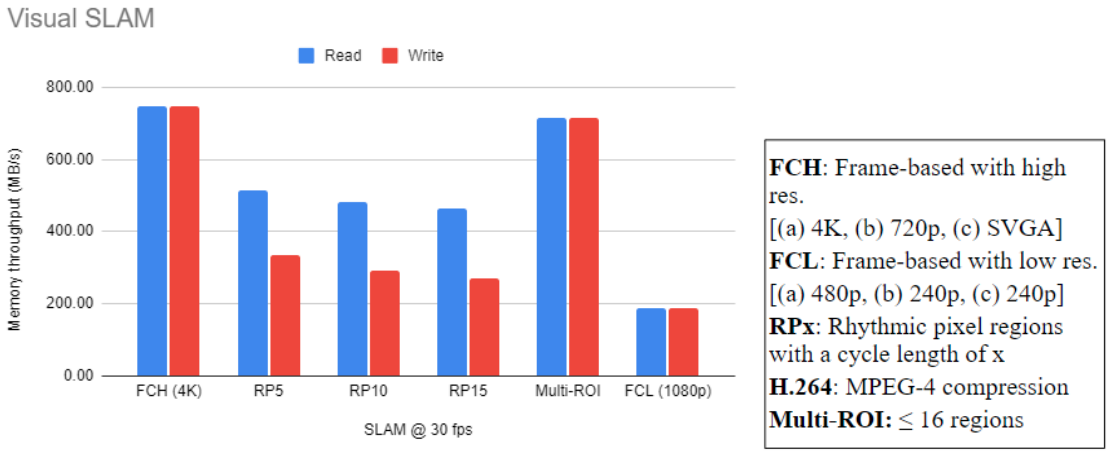
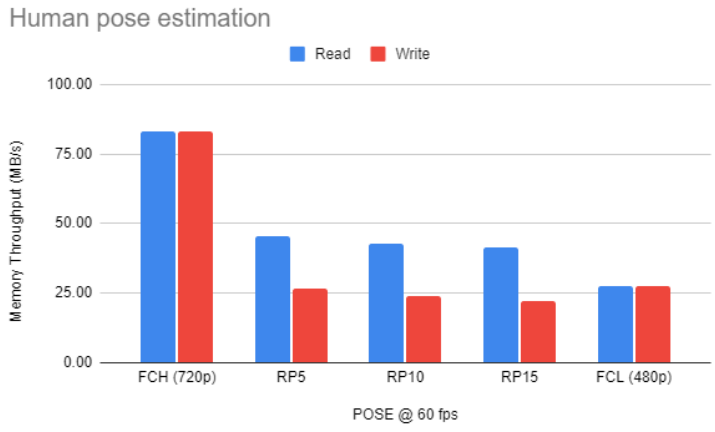


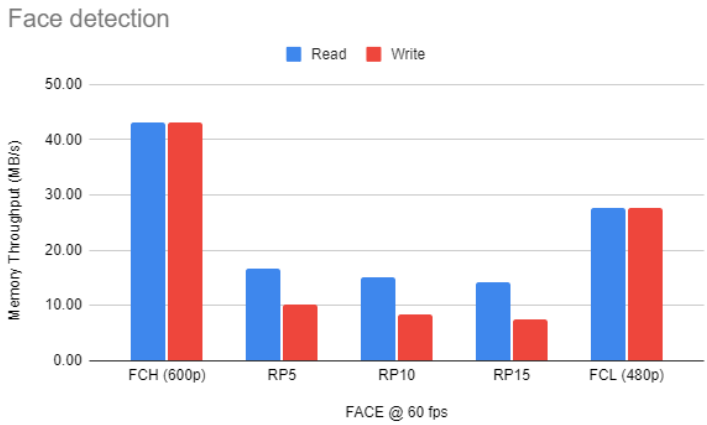
Figure 4 above showcases the potential throughput savings of rhythmic pixel regions when simulating frame-based computing and rhythmic pixel region-based computing on a Visual SLAM workload. FCH and FCL correspond to frame-based computing at high and low resolutions, respectively. RPx correspond to rhythmic pixel regions, where x denotes the cycle length. Multi-ROI corresponds to a simulation of a multi-ROI sensor system that has 16 regions of interest. Because multi-ROI supports less regions, many smaller regions had to be combined into a larger region, creating less memory efficient regions. The figure demonstrates rhythmic pixel regions ability to save memory throughput compared to both high resolution frame-based computing and current multi-ROI sensor implementations. In addition, despite providing task accuracy comparable to 4k resolution, rhythmic pixel regions use memory throughput roughly equal to two times as much as frame-based computing at 1080p.

Figure 5: Human Pose Estimation Throughput



As before in Figure 5, similar patterns arise when evaluating on a human pose estimation workload. Rhythmic pixel regions can use significantly less memory bandwidth compared to high resolution frame-based computing, and only slightly more memory bandwidth compared to low resolution frame-based computing.

Figure 6: Face Detection Throughput



Finally, in Figure 6, again similar patterns arise. However, in this case, rhythmic pixel regions can outperform frame-based computing at both high and low resolutions. This appears here because of the small relative resolution change between high resolution and low resolution frame-based computing, where high resolution is 600p, and low resolution is only 480p (compared to the larger jumps previously, such as from 4K to 1080p in Visual SLAM).

As shown by the figures, frame-based computing at full resolution always has a significantly higher memory throughput compared to rhythmic pixel regions with any cycle length configuration, as rhythmic pixel regions can discard a significant number of pixels. Currently, the region selection algorithm simply selects a region surrounding a point of interest determined by an algorithm, such as a face in face detection, and adds some padding to the point of interest before passing that region to the encoder. This allows consistent capture of important regions without transferring unnecessary pixels over the DDR interface. Additionally, rhythmic pixel region-based computing can reduce memory throughput significantly compared to multi-ROI computing due to the scalability and programmability of the encoder.

CHAPTER 5

CONCLUSION

In conclusion, frame-based computing is limiting the efficiency of visual computing algorithms by required a fixed and uniform resolution and framerate across an entire video sequence. I have developed a scalable and programmable I/O selection interface to selectively send important pixels, allowing for flexible spatiotemporal regions within a video sequence. The efficient encoding and decoding of the data allow significant reductions in memory throughput for common visual computing tasks with minimal modification required for the visual applications. This reduction in memory throughput allows efficient visual computing with high fidelity and high task accuracy where needed. In addition, I provided a rhythmic pixel region simulator to be used alongside memory traffic power simulators to estimate potential power savings of rhythmic pixel regions.

REFERENCES

1996scarlet. (n.d.). *faster-mobile-retinaface*. Retrieved from GitHub: https://github.com/1996scarlet/faster-mobile-retinaface

Andriluka, M. I. (2018). *PoseTrack: A Benchmark for Human Pose Estimation and Tracking.* CVPR.

CMU-SAFARI. (n.d.). *ramulator*. Retrieved from Github: https://github.com/CMU-SAFARI/ramulator

Jinhan Hu, J. Y. (2018). *Characterizing the Reconfiguration Latency ofImage Sensor Resolution on Android Devices.*

Raul Mur-Artal, J. M. (2015). *ORB-SLAM: a Versatile and Accurate Monocular SLAM System.* IEEE.

Saugata Ghose, A. G. (2018). *What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study.*

tukl-msd. (n.d.). *DRAMPower*. Retrieved from Github: https://github.com/tukl-msd/DRAMPower

Wikipedia. (n.d.). *High Efficiency Video Coding*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/High_Efficiency_Video_Coding

Xilinx. (n.d.). *Embedded Reference Platforms User Guide*. Retrieved from Github: https://github.com/Xilinx/Embedded-Reference-Platforms-User-Guide

Xilinx. (n.d.). *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. Retrieved from https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

Ximea. (n.d.). *Multiple ROI - Ximea*. Retrieved from Ximea: https://www.ximea.com/support/wiki/allprod/Multiple_ROI

Y. Wong, S. C. (2011). *Patch-based Probabilistic Image Quality Assessment for Face Selection and Improved Video-based Face Recognition.* IEEE.

Zhe Cao, G. H.-E. (2018). *OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields.* IEEE.