FALCON: Full Stack Evaluation Testbed of Compute Offloaded Extended Reality Systems

Zekai Lin

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-64
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-64.html

May 15, 2025

FALCON: Full Stack Evaluation Testbed of Compute Offloaded Extended Reality Systems

by Zekai Lin

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Yakun Sophia Shao

Professor Yakun Sophia Shao
Research Advisor

5/14/2025

(Date)

* * * * * * *

Chris Fletcher

Professor Christopher W. Fletcher
Second Reader

5/15/2025

(Date)

FALCON: Full Stack Evaluation Testbed of Compute Offloaded Extended Reality Systems

Zekai Lin

May 14, 2025

Abstract

This project explores full-stack software–hardware co-design for compute offloaded extended reality (XR) systems. Standalone VR headsets are increasingly preferred for their portability, but they face stringent power and performance constraints due to their limited hardware resources. To address this challenge, this work leverages ILLIXR, an open-source XR system testbed, in conjunction with Chipyard and FireSim to enable joint software and hardware evaluation. Three core contributions are presented. First, a graphics compute offloading system is implemented beneath the OpenXR abstraction in the Monado runtime, allowing VR applications to render on a host machine while streaming images to the headset. Second, a foveated video encoding pipeline is integrated using gaze estimation to reduce bitrate without loss of perceptual quality. Third, the visual-inertial odometry pipeline is accelerated using the RISC-V vector intrinsics and the Gemmini matrix accelerator. Evaluation demonstrates significant kernel-level speedups and highlights application-level bottlenecks, underscoring the importance of cross-layer optimization. This work establishes a foundation for future XR systems that balance efficiency and performance through domain-specific hardware and full-system co-design.

Acknowledgements

I would first like to thank Professor Yakun Sophia Shao for her invaluable support and guidance throughout this project. Her feedback was instrumental in shaping the direction and execution of my research.

I am also deeply grateful to Prashanth Ganesh, the PhD student I had the privilege of working with. He laid the foundation for the XR research that I was fortunate to join, and throughout the process, he provided crucial technical support, insights, and direction that significantly advanced the project.

Special thanks to Hansung Kim for welcoming me into his GPU research team and introducing me to the lab. Without his support, I would not have had the opportunity to participate in many exciting projects taking place there.

Finally, I would like to thank my parents for their financial and emotional support, without which this work would not have been possible.

Contents

List of Figures

List of Tables

List of Code Blocks

Chapter 1

Introduction

Extended Reality (XR) is steadily emerging as a mainstream computing platform with diverse applications in areas such as social interaction, education, healthcare, and robotics. Within the spectrum of XR devices, Virtual Reality (VR) headsets constitute the majority, encompassing both dedicated VR headsets and those equipped with Augmented Reality (AR) functionality via video see-through technology. VR headsets are typically classified into two categories: standalone headsets and PC-tethered headsets (PCVR).

PC-tethered VR headsets do not contain onboard processors and operate primarily as peripheral devices, comprising a display, audio system, and tracking sensors. These headsets connect to a personal computer via HDMI or DisplayPort for video output and USB for data transmission, relying entirely on the host system for processing tasks such as tracking, rendering, and audio. To meet the high bandwidth requirements of real-time rendering, graphics card manufacturers such as NVIDIA have introduced hardware and driver-level support, which allows stereo images to be transferred directly from GPU memory to the VR display.

In contrast, standalone VR headsets have integrated processors, such as Qualcomm's Snapdragon XR2, that allow the on-board execution of key workloads, including rendering, tracking, and video see-through. However, due to the performance limitations of mobile processors, a hybrid model has also emerged. In this configuration, VR applications are executed on a host PC, and the rendered stereo images are transmitted to the standalone headset via Wi-Fi or a wired connection.

Although PC-tethered VR headsets offer superior visual quality – benefiting from powerful desktop-grade graphics cards and effectively unlimited power budget – the industry's focus has increasingly been on the development of standalone VR headsets due to their portability and ease of use. However, standalone XR systems present numerous engineering challenges and design trade-offs. All computation is performed within the headset itself, which is constrained by a compact form factor and stringent power limitations. Rendering and tracking, the two most computationally intensive components, must operate at high frequencies. A typical target frequency for stereo image generation and head pose estimation is 90 Hz.

To maintain a lightweight and comfortable design, the standalone headsets are equipped with small batteries, which severely restrict the available power budget for real-time processing. As a result, optimizing XR systems for both performance and energy efficiency has become a significant area of research that demands co-design across both software and

hardware layers.

On the software side, advances include algorithms such as visual-inertial odometry for head tracking and variable-rate shading for efficient rendering. On the hardware side, custom accelerators, such as dedicated video codecs and machine learning accelerators, have been developed to improve computational throughput and energy efficiency. Although progress has been made independently in both domains, isolated optimizations in either software or hardware often fail to yield system-wide performance gains. Further improvements require holistic, cross-layer optimization that bridges software and hardware design.

Significant progress in numerical systems has been driven by both academic and open-source efforts in software and hardware. On the software side, ILLIXR (Illinois Extended Reality Testbed) supports system-level research through a modular architecture that enables benchmarking and evaluation of components such as visual inertial odometry, rendering, and audio (Huzaifa et al., 2022). Designed with a focus on user-perceived quality, ILLIXR integrates with the OpenXR standard and can operate as a driver in the Monado runtime, ensuring compatibility with XR applications. On the hardware side, Berkeley developed FireSim (Karandikar et al., 2018) and Chipyard (Amid et al., 2020) to advance domain-specific computing. FireSim provides high-fidelity FPGA-based hardware simulation, while Chipyard offers a flexible RISC-V-based SoC design framework for rapid prototyping of custom processors and accelerators. Together, they enable full-system evaluation by supporting execution of real workloads on simulated hardware, facilitating hardware-software co-design.

This project aims to bridge the gap between software optimization and hardware optimization for virtual reality platforms by using the ILLIXR software evaluation testbed alongside the Berkeley Chipyard framework. While ILLIXR provides a comprehensive environment for evaluating software performance, it is limited to the hardware platform on which it operates. By integrating Chipyard, this work enables full stack optimization of both the virtual reality system software and the underlying hardware architecture.

This study represents an initial step toward full software and hardware co-design for virtual reality systems, with a focus on the two most computationally demanding components: rendering and tracking. The research introduces three primary contributions:

1. The development of a compute offloading infrastructure that moves graphics rendering workloads from VR headsets to host machines.

2. The integration of foveated video encoding techniques using gaze prediction from eye tracking.

3. The acceleration of visual inertial odometry algorithms using vector processing and specialized hardware units from the Chipyard framework.

Chapter 2

Background and Motivation

2.1 ILLIXR System

The ILLIXR system integrates three primary pipelines, perception, visual, and audio, that together encompass the core components found in modern XR systems. The perception pipeline includes a visual-inertial odometry (VIO) system for headset localization, using both camera and IMU data. ILLIXR leverages the OpenVINS library (Geneva et al., 2020) to estimate headset pose, which can then be queried by the rendering application to generate stereo images. The visual pipeline is responsible for post-processing rendered images prior to display. Specifically, it performs asynchronous reprojection to correct for motion between the time of rendering and the time of display, improving visual stability. The audio pipeline manages spatial audio but is not examined further in this project. This project introduces several extensions to the ILLIXR system. First, the visual pipeline is modified to enable graphics compute offloading, simulating a wireless VR streaming scenario in which frames are rendered on a host machine and transmitted to an edge VR headset. Second, foveated video encoding is implemented using gaze prediction from eye-tracking data to improve streaming efficiency. Finally, the VIO modules in the perception pipeline are analyzed and optimized through hardware–software co-design targeting a Chipyard SoC platform that represents a standalone VR headset.

The ILLIXR system is designed as a modular framework in which each XR system component is implemented as a dynamically loadable plugin. Shown in Figure 2.1, this architecture allows researchers to interchange modules to evaluate system-level performance under different configurations. Communication between plugins follows a producer-consumer model, supporting both synchronous (blocking) and asynchronous (non-blocking) message passing. This research extends the ILLIXR system by integrating a graphics compute offload module and foveated video decoding modules, enabling evaluation of wireless VR streaming with gaze-adaptive rendering.

The ILLIXR system was benchmarked on three different hardware platform configurations, one high-end desktop platform, and NVIDIA Jetson AGX Xavier platform with either maximum clock frequencies and half clock frequency. The high end desktop platform simulated the PCVR setting where the headset is tethered to a PC where all computation happens. The Jetson platform simulates a standalone VR platform where all computation happens in a standalone VR headsets with limited power consumption and small form factor. Benchmark-

ing results indicate that, across the full XR system, including the application layer running atop ILLIXR, the application logic and the visual inertial odometry pipeline are the most computationally intensive components. Notably, the application's computational demand varies significantly with the complexity of the rendered scene, highlighting the impact of rendering workload on overall performance.

These benchmark results motivate the approach taken in this project. At the application level, we aim to offload the graphics workload to a host machine, rather than executing it on the edge VR headset. This not only frees up compute resources on the headset for VIO, but also enables the development of more complex applications, such as medical training, virtual conferences, and high-fidelity games, that demand higher visual quality. Additionally, since VIO remains one of the most compute-intensive components on the edge device, it calls for careful hardware–software co-design to achieve both high performance and power efficiency.

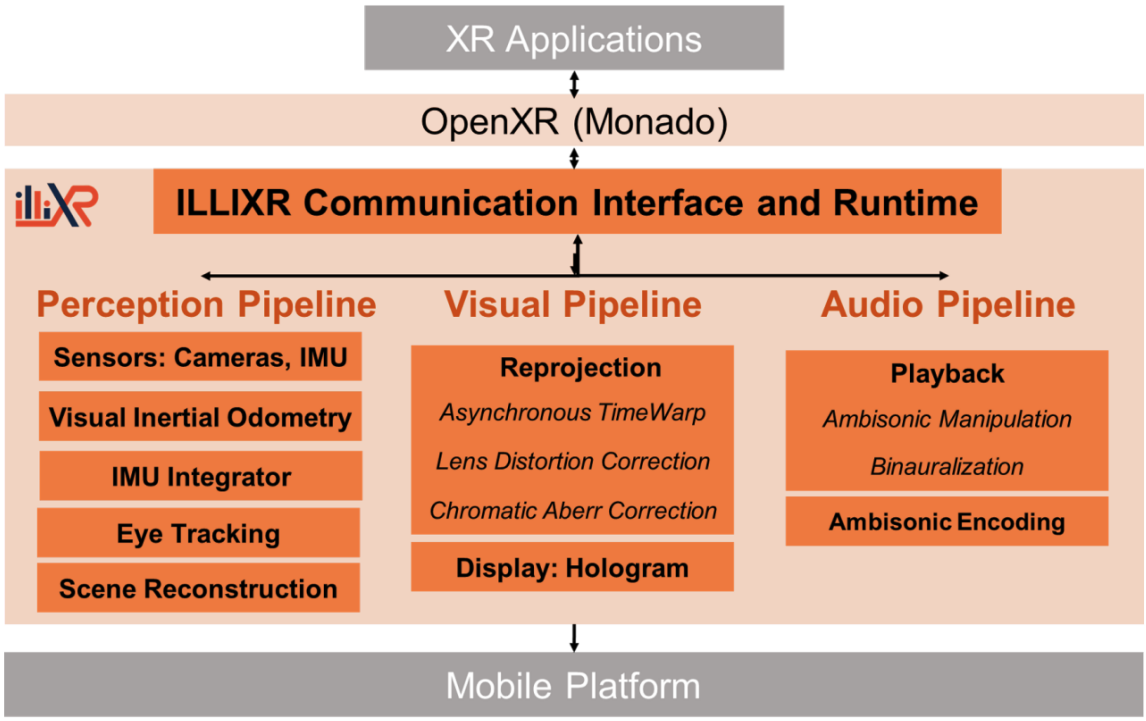

Figure 2.1: The ILLIXR system.

2.2 Monado Runtime

OpenXR has become the industry standard for enabling XR applications to interface with the low-level runtime systems that drive virtual and augmented reality headsets. Shown in Figure 2.2, it provides a unified set of APIs for creating XR sessions, allowing applications to interact with XR devices in a synchronized manner using functions such as `xrWaitFrame`, `xrBeginFrame`, and `xrEndFrame` for frame pacing. Within each frame, applications can query headset poses, access controller inputs, and submit rendered frames using graphics APIs,

such as Vulkan or OpenGL. OpenXR enables applications to implement custom rendering pipelines while offering a standardized interface for accessing tracking data and user input. This abstraction allows major game engines to extend their backends to support any XR device compliant with the OpenXR specification.

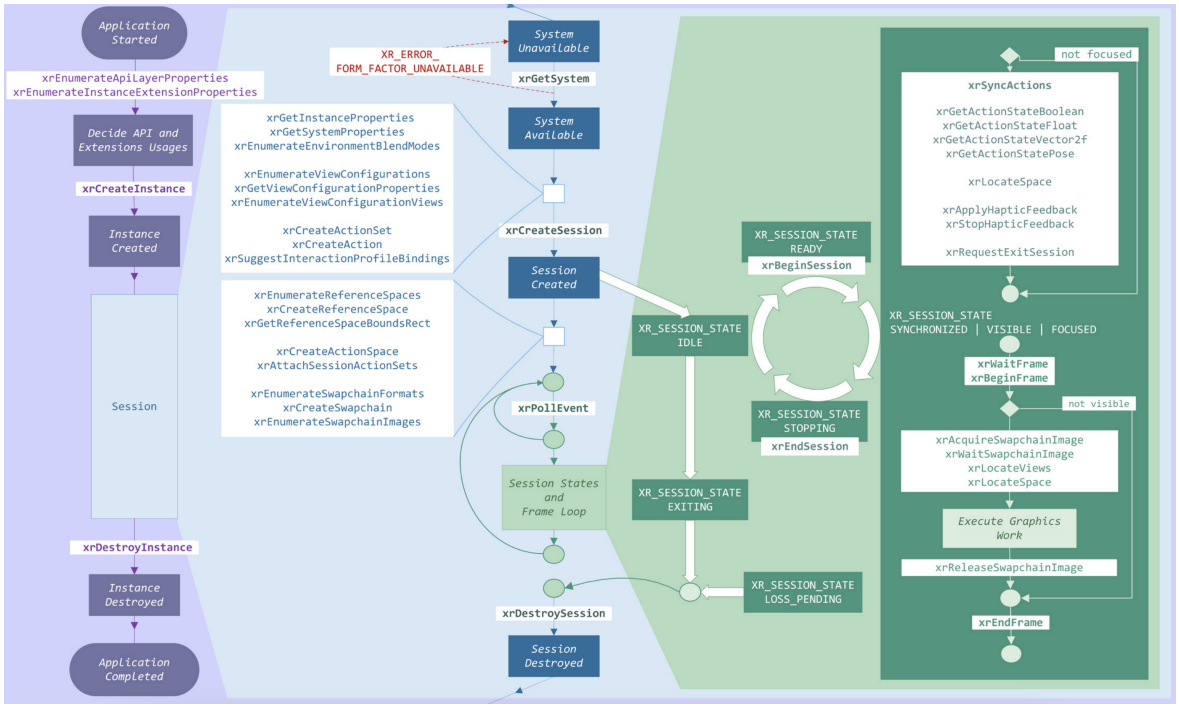

Figure 2.2: The OpenXR Standard.

Monado is an open-source runtime that implements the OpenXR standard, providing a flexible and extensible platform for interfacing with XR hardware. It supports the integration of hardware-specific drivers that communicate directly with VR headsets. During initialization, Monado scans all available drivers to identify and select the connected VR device. The corresponding device driver is then used to handle communication with the hardware. Implementing a new driver is relatively straightforward, requiring the developer to define interface functions for retrieving headset pose and controller input data. When an OpenXR application requests such data, Monado forwards the query to the appropriate driver implementation, executes the relevant callback, and returns the results to the application. When the application submits rendered frames via OpenXR, Monado processes them in its compositor. Typical post-processing tasks done in the compositor include merging stereo images from the left and right eyes, applying asynchronous timewarp, and compositing frames submitted by multiple OpenXR applications. The final images are then transferred directly to the VR headset display through the GPU.

While Monado provides robust support for tethered VR headsets, it does not natively support compute offloaded XR systems, in which all XR system components, except for the application, execute on a standalone headset. In this project, Monado is extended to support such a compute offload architecture. A custom driver is implemented to enable communication between Monado and the ILLIXR system, which can run either as an independent process

or within an FPGA simulation environment. This modification allows for hardware-software co-design of compute offloaded XR systems, where the application logic is offloaded to a host computer. The approach is compatible with any OpenXR application, facilitating comprehensive performance evaluation of the system. Furthermore, by executing ILLIXR within an FPGA simulation, both software and hardware components can be jointly optimized.

2.3 Visual Inertial Odometry

Visual inertial odometry is a technique used to estimate the position and orientation of a device by combining input from a camera and an inertial measurement unit (IMU). By integrating visual features with measurements of acceleration and angular velocity, VIO enables accurate tracking of motion over time, providing reliable localization in dynamic environments. Unlike simultaneous localization and mapping (SLAM), VIO does not construct a global map, making it more lightweight and well suited for real-time applications such as augmented and virtual reality, as well as drone navigation. VIO systems are typically composed of two main components: the frontend and the backend. The frontend processes image data to extract visual feature points, while the backend fuses these features with inertial data from the IMU to estimate the device's pose, including both its position and orientation.

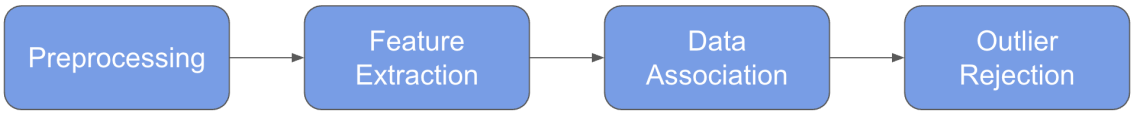

Figure 2.3: Visual inertial odometry frontend.

The frontend of a VIO system is typically more lightweight than the backend, with a relatively consistent computational cost across image frames. Figure 2.3 shows the frontend is composed of four primary stages: preprocessing, feature extraction, data association, and outlier rejection. Preprocessing prepares raw image data obtained from the camera sensor for subsequent analysis. This stage often includes converting images to grayscale, applying histogram equalization to improve global contrast, and constructing image pyramids to support optical flow estimation at multiple scales. Feature extraction identifies salient and repeatable visual features, such as corners or edges, from the preprocessed images. Common algorithms include FAST (Features from Accelerated Segment Test) (Rosten and Drummond, 2005) and the Harris corner detector (Harris and Stephens, 1988). Data association matches corresponding feature points across multiple camera frames, enabling the tracking of visual landmarks over time. This step is critical for establishing temporal coherence in motion estimation. Algorithms such as BRIEF (Binary Robust Independent Elementary Features) (Calonder et al., 2010) are commonly used in this stage. Outlier rejection is the final stage in the frontend pipeline. Since nearest neighbor matching can be error prone, it is necessary to eliminate incorrect associations. The filtered set of feature points and their correspondences across frames are passed from the frontend to the backend, where they are fused with inertial data to estimate the device's pose.

The backend of a visual inertial odometry system is responsible for estimating the pose of the device, which includes both its rotation and position, by fusing visual features from

Algorithm	Cost Construction	Gauss-Newton	Marginalization
EFK	Current pose and all current and past features	One	All past poses
MSCKF	Current IMU state and n past poses, evenly spaced in time	One	All other poses
PGO	Current and all past poses, with no features	Multiple, until convergence	None

Table 2.1: A list of VIO backend algorithms.

the frontend with inertial measurements from the IMU. This process is formulated as an optimization problem that seeks to compute the most likely trajectory of the device over time (Saxena et al., 2022). Common approaches include filtering techniques such as the Kalman filter and its variants. As shown in table 2.1, notable backend algorithms include the Extended Kalman Filter (EKF), the Multi-State Constraint Kalman Filter (MSCKF) (Mourikis and Roumeliotis, 2007), and pose graph optimization (PGO). Among these, MSCKF is widely adopted in real-time applications due to its balance between accuracy and computational efficiency.

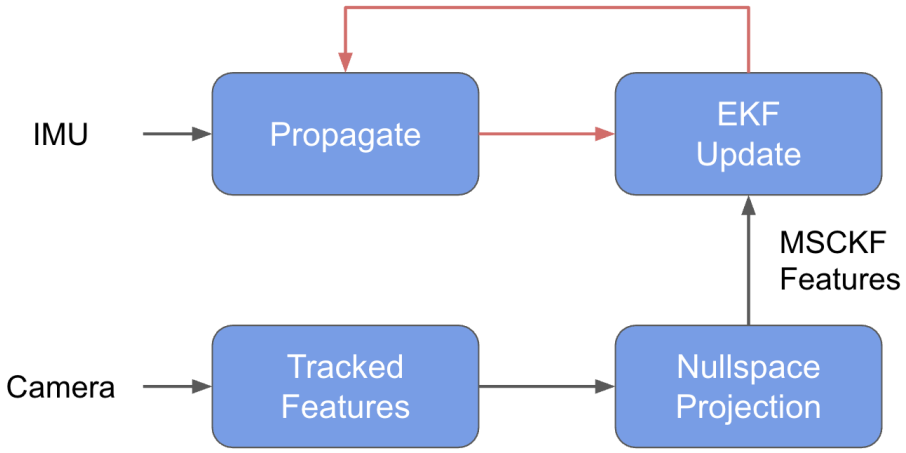

Figure 2.4: An illustration of Multi-State Constraint Kalman Filter.

Kalman filter based methods typically consist of two key components: the propagation stage (also referred to as the prediction stage) and the update stage (also known as the correction stage). Taking the Multi-State Constraint Kalman Filter shown in Figure 2.4 as an example, during the propagation stage, inertial measurements from the gyroscope and accelerometer are used to estimate the device's state over time. This is accomplished by integrating the motion dynamics forward through discrete time steps, yielding predicted pose estimates. However, these propagated poses are only predictions and must be refined using visual measurements in the update stage. In this stage, a measurement model is employed to

capture the geometric constraints imposed by observing static visual features from multiple camera perspectives. The update stage then corrects the predicted state by minimizing the reprojection error, the difference between the observed feature positions provided by the frontend and the projected feature positions predicted using the current pose estimate.

2.4 Chipyard and FireSim

Berkeley has developed two important infrastructures to accelerate research in domain specific computing: FireSim and Chipyard. FireSim is an FPGA based platform for full system hardware simulation, enabling high fidelity evaluation of processor and system designs. Chipyard is a chip design framework built on the RISC-V architecture that allows hardware researchers to rapidly iterate on processor and accelerator designs. Chipyard supports fast prototyping of customized hardware by allowing developers to add or remove accelerator components such as the Saturn vector unit (Zhao et al., 2024) or the Gemmini systolic arrays (Genc et al., 2021). It also supports switching between different processor cores, including the five-stage in-order Rocket core (Asanović et al., 2016), the Berkeley out-of-order machine (BOOM) (Zhao et al., 2020), and a six-stage dual issue core called Shuttle. FireSim complements Chipyard by enabling full application-level workloads to be executed on the simulated hardware design.

In this project, Chipyard and FireSim are integrated with ILLIXR to enable hardware software co-design for extended reality systems. The ILLIXR framework is executed on a custom RTL design simulated via FireSim, which supports full-system evaluation, including booting a Linux-based operating system and running XR workloads directly. This setup extends system evaluation beyond software-only configurations by allowing researchers to modify both software modules within ILLIXR and hardware components within Chipyard. The project specifically targets the visual-inertial workload, identified as the one of the most computationally intensive tasks in XR systems, for benchmarking and evaluation. Various kernel backends, such as vector processing units and matrix computation accelerators implemented in Chipyard, are tested to assess their performance impact on the end-to-end system.

Chapter 3

Implementation

Figure 3.1 presents the architecture of the compute offloaded XR system. The system enables offloading of graphics workloads by streaming rendered frames, such as those generated by the open-source Filament renderer, from a host machine to a standalone VR headset, while maintaining real-time feedback of pose and gaze. To ensure compatibility with all OpenXR-compliant applications, the offloading mechanism is implemented beneath the OpenXR abstraction layer within the Monado runtime. Monado captures rendered images using Vulkan, encodes them via the x264 implementation in FFmpeg, and streams the data to the ILLIXR system over a dedicated link. On the headset side, ILLIXR decodes the video stream, performs eye tracking using RITnet, and executes visual-inertial odometry to estimate the user's pose. Gaze information is used for foveated video encoding, allowing bandwidth reduction without significantly degrading visual quality. The architecture supports deployment of ILLIXR either natively on a host machine or on a simulated system-on-chip using FireSim within the Chipyard framework, enabling full-stack system evaluation and hardware-software co-design.

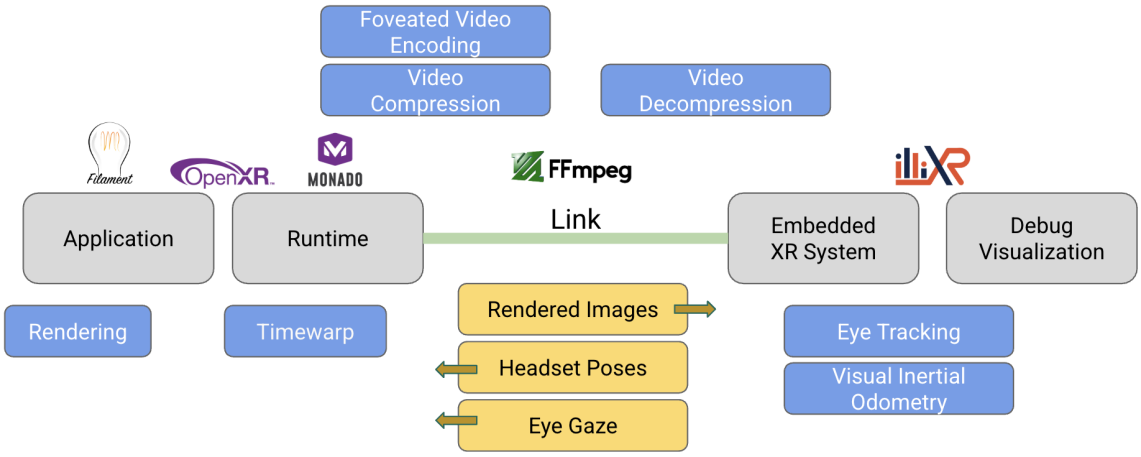

Figure 3.1: System architecture.

3.1 Graphics Compute Offload

The objective of graphics compute offloading is to enable application workloads to be executed on a host computer, while the rendered stereo images are streamed to an edge VR headset. To ensure compatibility with all OpenXR-compliant applications, the compute offload functionality is implemented below the OpenXR abstraction layer. This design allows for full-stack system benchmarking with a wide range of XR applications, including those built with game engines such as Unity, Unreal Engine, and Godot. The implementation consists of three main components: modifications to the Monado runtime, the development of a streaming interface between Monado and ILLIXR, and the creation of an ILLIXR plugin capable of receiving and processing the streamed image data.

3.1.1 Monado Compute Offload Driver

On the Monado side, a compute offload driver is implemented to support both pose querying from the remote VR headset and streaming of image data back to the headset. Handling pose data is relatively straightforward, as Monado already provides an interface for retrieving pose information from device drivers. When Monado invokes the driver callback, the function sends a pose request to the remote headset and returns the received pose to the Monado runtime, which subsequently forwards it to the VR application. To optimize performance, the pose retrieval mechanism is designed to be asynchronous. If the receive (RX) queue has not yet obtained the latest pose from the headset, the previously cached pose is returned instead. Whenever a new pose is received, it is stored in the cache. This non-blocking design eliminates network-induced latency from the application rendering pipeline, at the cost of occasionally using slightly outdated pose data, a trade-off considered acceptable for VR applications that require high update frequencies.

In contrast, Monado lacks native support for streaming image data, as it is originally designed to map rendered images directly from GPU memory to a tethered VR display. To enable image streaming to a remote headset, rendered image data must be extracted from GPU memory using the Vulkan API. Specifically, modifications are made to the compositor component of the Monado runtime. Before an image in the swapchain is presented to the display, it is transferred back to system memory using `vkCmdCopyImageToBuffer`. The resulting buffer is then passed to the compute offload driver, which transmits the image data to the VR headset over a dedicated communication link.

3.1.2 Communication Link

The communication link between the Monado runtime and the ILLIXR system is designed to support a wide range of deployment scenarios, including wireless streaming over a local area network (LAN) and simulation-based communication via FireSim between a host computer and a simulated SoC. To ensure compatibility and extensibility, the data transmission format shown in code block 3.1 is designed to conform to RoSE (Nikiforov et al., 2023), an open-source hardware-software co-simulation framework that enables full-stack, pre-silicon, hardware-in-the-loop evaluation. RoSE uses a TCP socket interface with a defined message format: each message begins with a header that specifies the command type and the size of the payload, if applicable. When the payload size is nonzero, the payload immediately

follows the header in the bitstream. Within the compute offload driver in the Monado runtime, a TCP client is implemented according to this specification. This client can establish connections with the RoSE interface or with any compatible TCP server running on a device within the same network.

```
typedef struct header
{
    uint32_t command;
    uint32_t payload_size;
} header_t;

typedef struct message_packet
{
  header_t header;
  char *payload;
} message_packet_t;
```

Listing 3.1: Message format used for pose and image streaming.

3.1.3 ILLIXR Compute Offload Plugin

On the ILLIXR side, a custom plugin is developed to serve as the communication endpoint with the host system. When ILLIXR is executed natively on the host machine, this plugin functions as a TCP server that communicates with the compute offload driver in Monado using the defined protocol. In scenarios where ILLIXR runs within the FireSim simulation environment, the plugin instead interfaces with the external system through the Direct Memory Access (DMA) bridge provided by the RoSE framework. Within the plugin, when a pose request is received from the host, it queries a pose published by another plugin in the ILLIXR system. When image data is received, it is written to a buffer, and a handle to this buffer is published to the rest of the ILLIXR pipeline, enabling downstream components to access the received image for further processing or display.

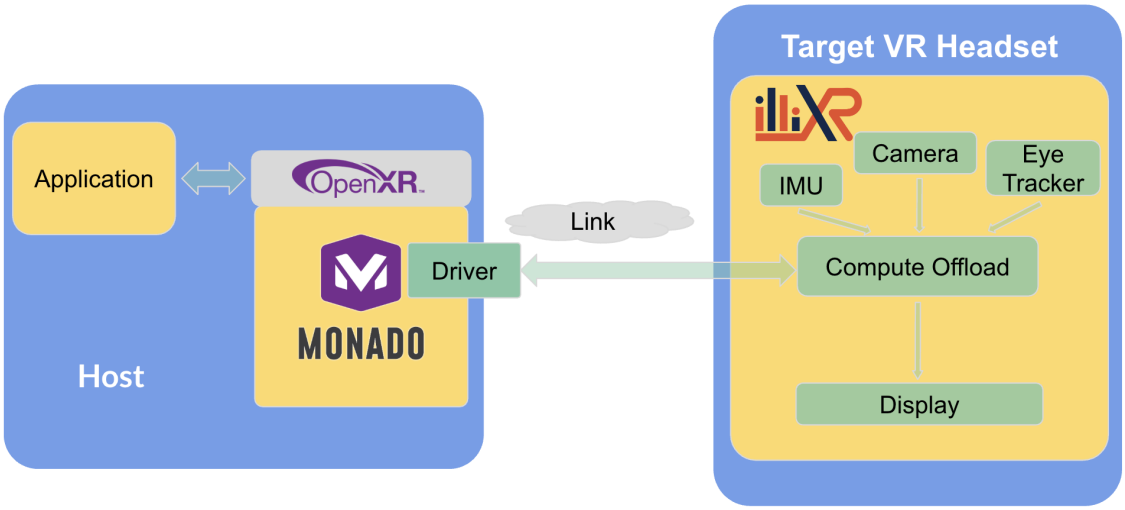

Figure 3.2: An illustration of the graphics compute offload system.

Figure 3.2 illustrates the resulting system architecture. On the host side, an OpenXR application interfaces with the Monado runtime via the standard OpenXR API. Monado operates as an independent process and establishes an inter-process communication channel with the OpenXR application. Through this interface, Monado receives rendered images from the application and transmits head-mounted display (HMD) pose data back to it. Additionally, Monado loads the compute offload driver, which establishes a communication link with the ILLIXR system. The ILLIXR system receives emulated camera and IMU data to perform localization using visual-inertial odometry. It also maintains a connection to the Monado runtime, over which it receives streamed video frames and returns updated HMD pose data, enabling a closed-loop interaction between the offloaded graphics application and the headset-side pose estimation.

3.2 Foveated Video Encoding

3.2.1 Eye Tracking

Eye tracking is implemented using RITnet from Chaudhary et al. (2019). RITnet is a lightweight, fully convolutional neural network designed for real-time semantic segmentation of the eye region, identifying the sclera, iris, pupil, and background. Its encoder-decoder architecture with skip connections ensures accurate segmentation while maintaining low computational demands, making it suitable for deployment on edge devices and mobile platforms. In the project, the RITnet is exported to a format supporting ONNX runtime. A library is built for inference tasks that compute the gaze position from images of near-eye cameras. The library is then integrated into the ILLIXR system as a plugin running on a separate thread. For each iteration of the loop, it makes two inferences, one for each eye. The computed gaze results are then published to the host side through a TCP socket for foveated video encoding.

3.2.2 Video Codec

FFmpeg is a robust multimedia framework widely utilized for video encoding and decoding, with x264 being one of its most prominent codecs for encoding video into the H.264 format. x264 is an open-source software library that delivers highly efficient video compression while preserving visual quality at reduced bitrates, making it an optimal choice for VR image streaming. x264 implements the H.264 video compression standard through a series of stages shown in the Figure 3.3. Initially, video frames are processed and divided into macroblocks for detailed analysis. The codec employs intra-frame prediction to achieve compression within a single frame by predicting blocks based on neighboring pixel data. Additionally, inter-frame prediction compresses data across successive frames by identifying changes through motion estimation and motion compensation. Following prediction, the data is transformed into coefficients, which are subsequently subjected to quantization, the primary stage where compression occurs. As shown in Figure 3.4, x264 has a rate control mechanism that adjusts the amount of compression on each macroblock based on some conditions. A quantization parameter is assigned to each macroblock, as determined by the selected rate control algorithm, effectively regulating the level of compression applied.

In this project, we use Constant Rate Factor (CRF) for rate control. CRF is a quality-based rate control mode in x264, where the encoder dynamically adjusts the bit rate to

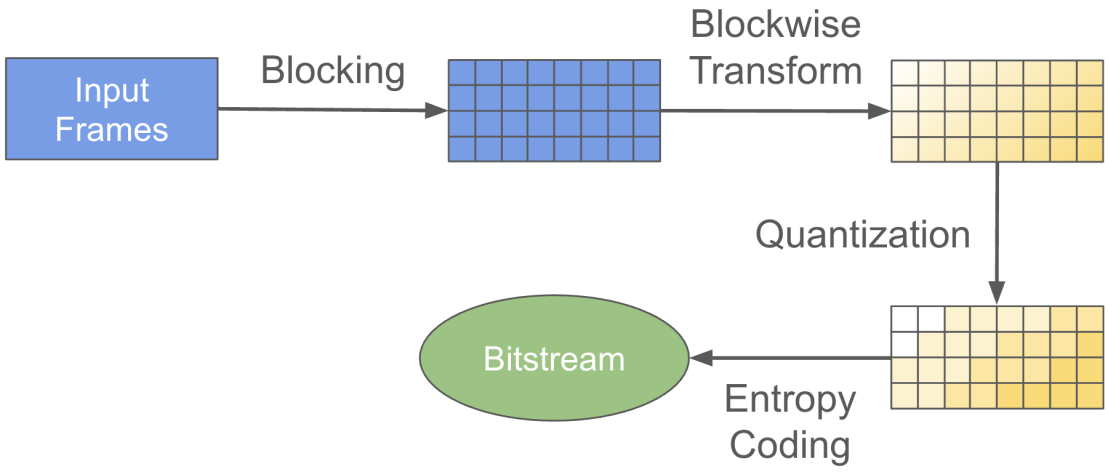

Figure 3.3: Compression algorithm.

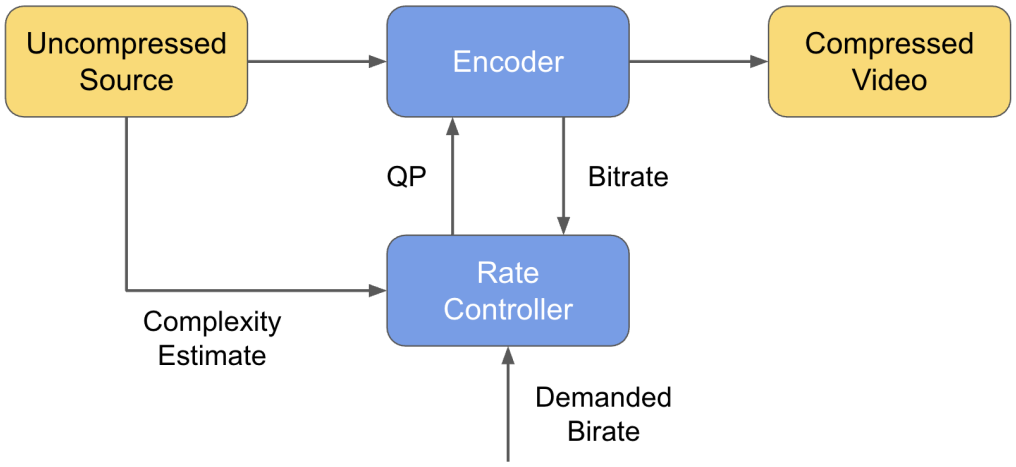

Figure 3.4: Rate control scheme.

maintain consistent visual quality across frames. It operates on a scale of 0 to 51, with lower values resulting in higher quality and larger file sizes. A CRF value of 28 was selected as it provides a reasonable balance between visual quality and bit rate. The process begins with transferring the image in RGBA format from the GPU to the CPU using Vulkan. The RGBA image is subsequently converted to the YUV422P format, which is supported by the x264 library for video encoding. The images in YUV422P format are then encoded to an H.264 bitstream and transmitted from the host Monado system to the standalone headset, represented by the ILLIXR system. On the ILLIXR side, the received H.264 bitstream is decoded back into RGBA images, completing the pipeline. Figure 3.5 gives an overview of the pipeline.

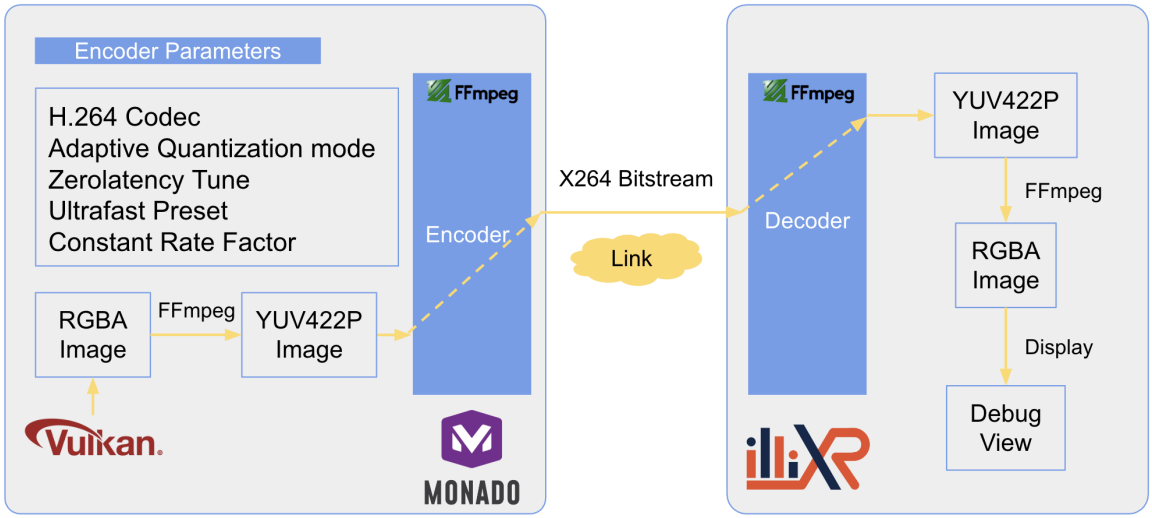

Figure 3.5: Video compression pipeline.

3.2.3 Foveated Encoding

Foveated video encoding is built on top of the x264 library that FFmpeg uses. X264 library supports specifying a quantization offset to the quantization parameter for each macroblock. This allows the users to have additional control over the degree of quantization on top of the rate control algorithms. For foveated video encoding, we adopt the idea from Illahi et al. (2020) and Wiedemann et al. (2020), and calculate the quantization offset for each macroblock using the following equation:

$$\bar{q}(x,y) = \delta\left(1 - \exp\left(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right)\right) \tag{3.1}$$

The given equation defines a two-dimensional Gaussian function used to model visual quality distribution in foveated video encoding. The center of the Gaussian, denoted by (x_0, y_0), corresponds to the user's gaze position, where the highest visual fidelity is maintained. Quality degrades smoothly with increasing distance from this focal point, governed by the standard deviation σ, which controls the rate of falloff. The scaling factor δ specifies the

maximum reduction in quality at the periphery of vision, enabling bitrate savings while preserving perceptual quality in virtual reality applications.

To implement this approach, how FFmpeg interfaces with the x264 library needs to be modified. In particular, the Region of Interest (ROI) feature in FFmpeg, which utilizes x264's quantization offset API, is modified to compute quantization offsets based on the Gaussian function described above, using the gaze position, σ, and δ. Figure 3.6 illustrates the distribution of quantization offsets applied across the image. Since the implementation continues to rely on the standard x264 API, the resulting encoded images remain compliant with the H.264 codec specification, eliminating the need for any modifications on the decoder side to support foveated video encoding.

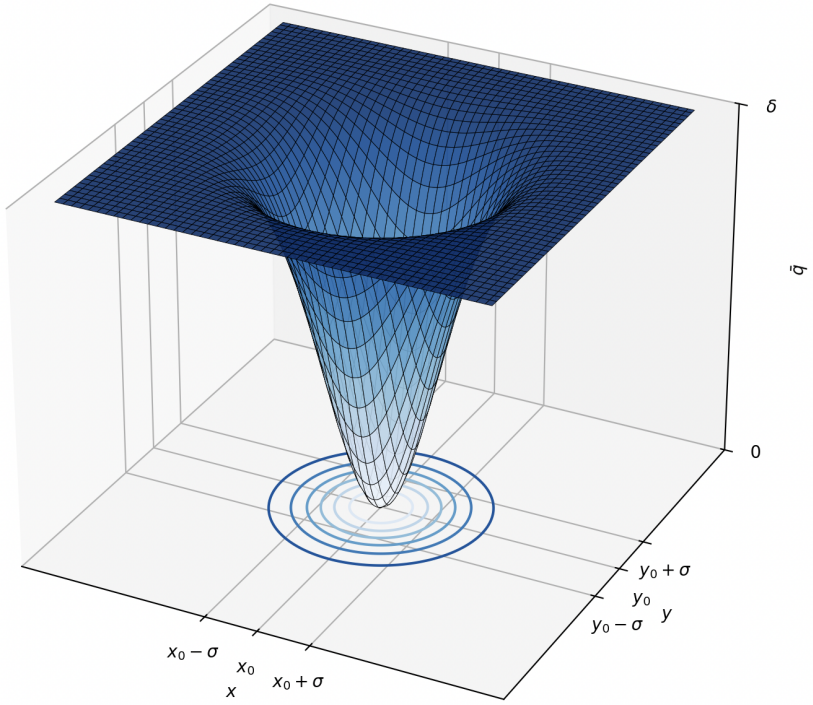

Figure 3.6: Encoding function.

3.3 Visual Inertial Odometry

To enable acceleration of state estimation on custom hardware, several modifications are required in the OpenVINS framework. OpenVINS relies on the Eigen3 library for all back-end computations. Eigen3 supports integration with custom linear algebra kernels through the Basic Linear Algebra Subprograms (BLAS) interface. In this project, an open source linear algebra library named OpenBLAS is used to serve as the backend for Eigen's matrix operations (Wang et al., 2013). This setup allows for the substitution or extension of computational kernels with custom implementations that leverage RISC-V vector instructions or specialized hardware accelerators. Figure 3.7 illustrates the software dependencies and

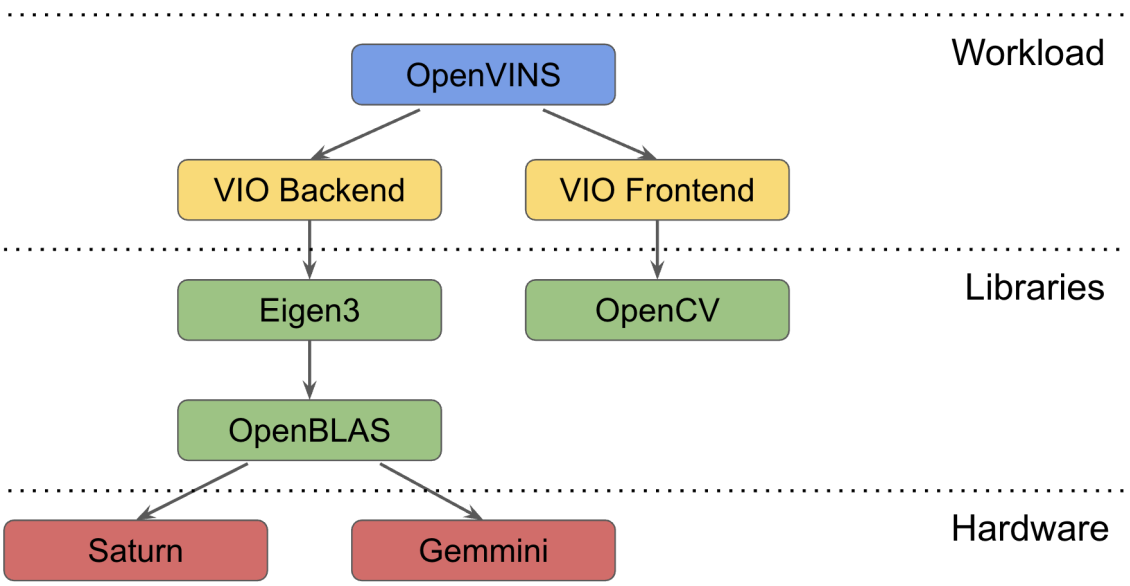

Figure 3.7: OpenVINS software stack and hardware accelerators.

optional hardware acceleration backends used by the VIO backend of the OpenVINS library. OpenBLAS includes preliminary support for the RISC-V architecture through three distinct backends: a scalar implementation and two vector implementations targeting vector lengths (VLEN) of 128 and 256 bits, respectively. However, these backends are outdated, and the kernels require significant reimplementation to achieve good performance.

3.3.1 Vector Backend Implementation

The GEneral Matrix–Matrix multiplication (GEMM) kernels are optimized using RISC-V vector intrinsics. The vectorized implementation adopts principles from Goto's BLAS approach, wherein submatrices of matrices A and B are remapped into scratch buffers to improve cache efficiency (Goto and Geijn, 2008). The vector kernels then operate directly on these scratch buffers.

```
1     for(BLASLONG k=1; k<K; k++) {
2         B0 = B[bi+0];
3         B1 = B[bi+1];
4         B2 = B[bi+2];
5         B3 = B[bi+3];
6         bi += 4;
7
8         A0 = __riscv_vle32_v_f32m2( &A[ai+0*gvl], gvl );
9         ai += 8;
10
11        result0 = __riscv_vfmacc_vf_f32m2( result0, B0, A0, gvl);
12        result1 = __riscv_vfmacc_vf_f32m2( result1, B1, A0, gvl);
13        result2 = __riscv_vfmacc_vf_f32m2( result2, B2, A0, gvl);
14        result3 = __riscv_vfmacc_vf_f32m2( result3, B3, A0, gvl);
```

```
15      }
```

Listing 3.2: Single precision GEMM kernel on vector units with register length of 128 bits.

On hardware with 128-bit vector registers, the backend configured for `vlen=128` is utilized. The single-precision GEMM kernel computes an 8×4 submatrix of matrix C per invocation, aiming to maximize the utilization of available vector registers. Intermediate results are stored in variables of type `vfloat32m2_t`, which span two vector registers and can store eight single-precision floating-point values. The kernel's core computation is structured as a triple-nested loop that performs matrix multiplication. Prior to entering the innermost loop, eight elements from a column of the A submatrix are loaded into a vector register using the `_riscv_vle32_v_f32m2` intrinsic. Simultaneously, four elements from the first row of the B submatrix are loaded into scalar registers and multiplied with the vector register using the `_riscv_vfmul_vf_f32m2` intrinsic. The resulting intermediate values are stored across four vector registers. The innermost loop, illustrated in code block 3.2, employs the fused multiply-accumulate intrinsic `_riscv_vfmacc_vf_f32m2` to compute dot products. In each iteration, four scalar elements from a row of the B submatrix are loaded, each multiplied with its corresponding vector register, and the products are accumulated into the output vectors.

Data Type	**128-bit Vector Registers**	**256-bit Vector Registers**
Single Precision	8x4 Kernel Size	16x8 Kernel Size
Double Precision	8x4 Kernel Size	8x8 Kernel Size

Table 3.1: Kernel size configuration

For hardware equipped with 256-bit vector registers, a distinct kernel is employed for single-precision GEMM. With the increased register width, this kernel computes larger submatrices of C, specifically of size 16×8. For double-precision GEMM operations, two specialized kernels are developed to support both 128-bit and 256-bit vector register configurations. The table 3.1 shows the kernel size configuration for different hardware backends and data types. While the general strategy for matrix multiplication remains consistent across different hardware configurations and data types, the key distinction lies in the amount of data each kernel processes per iteration, based on vector register capacity. Tail cases, which handle matrix dimensions not divisible by the block size, are also managed differently depending on the specific kernel implementation.

For the GEneral Matrix-Vector multiplication (GEMV) operation, the RISC-V Vector kernel implementation requires no modification. Two versions of the GEMV kernel are provided: one for the standard layout and another for the transposed matrix case. Due to the simplicity of the implementation, both the 128-bit and 256-bit vector register configurations utilize a common backend. The kernel maximizes utilization of the vector registers by loading as many elements as possible and processing them within a nested loop structure.

3.3.2 Gemmini Backend Implementation

Gemmini is an open-source, full-stack generator of deep neural network (DNN) accelerators integrated within the Chipyard framework. It serves as a hardware accelerator capable of

executing fundamental operations such as matrix multiplication, convolution, and residual addition. In this work, Gemmini's matrix multiplication functionality is leveraged to implement GEMV and GEMM operations within OpenBLAS. Unlike the RISC-V Vector (RVV) backend, the Gemmini backend is architecturally distinct. Gemini hardware includes a dedicated runtime that employs heuristics to manage data movement into its local scratchpad memory. This design introduces compatibility challenges with OpenBLAS, which also employs its own data management strategy to move submatrices into cache-friendly scratchpad buffers. Since OpenBLAS invokes low-level kernels through its runtime, it is responsible for loading the working set into buffers prior to computation. Consequently, both Gemmini and OpenBLAS attempt to manage data movement independently, leading to potential redundancy. Directly replacing the RVV kernels with Gemmini's low-level matrix multiplication interface would result in duplicated data transfers and buffer allocations, incurring significant performance overhead.

Therefore, rather than integrating the Gemmini hardware accelerator at the backend kernel level of OpenBLAS, a more appropriate approach is to invoke the Gemmini runtime at the BLAS interface level. At this level, the arguments for GEMM and GEMV operations are captured directly. Instead of passing these arguments to the OpenBLAS runtime, they are redirected to the Gemmini runtime, allowing Gemmini to manage all aspects of data movement and execute its own tiled matrix multiplication. However, this translation is nontrivial. OpenBLAS assumes a column-major memory layout, whereas the Gemmini runtime operates on row-major matrices. In addition to this mismatch, the two runtimes differ in their parameter conventions and configuration options. The OpenBLAS GEMM interface is shown in code block 3.3. For example, OpenBLAS supports matrix transposition, variable stride sizes, and scaling by scalar factors, all of which must be correctly translated to their equivalents in the Gemmini interface. Addressing these discrepancies requires careful adaptation to ensure correctness and performance.

Another limitation of using the Gemmini backend is that the Gemmini hardware only supports single-precision floating-point operations and lacks native support for double precision. However, the visual-inertial workload implemented using the OpenVINS library performs all computations in double precision. To enable Gemmini to support GEMM and GEMV operations on double-precision data, the only viable approach is to cast the matrix elements from double to single precision before passing them to the Gemmini runtime. To minimize the overhead of this casting process, only the elements actually used in the matrix multiplication are converted. Specifically, elements in buffer regions that are not accessed, due to stride sizes exceeding the width or height of the matrix, are excluded from casting. For example, in a row-major matrix of width N with a stride greater than N, only the first N elements of each row are cast to single precision. During testing, the output produced by the Gemmini backend is printed to the console and compared against the results generated by the native OpenBLAS implementation to verify correctness.

```
1 void CNAME(
2     enum CBLAS_ORDER order,
3     enum CBLAS_TRANSPOSE TransA, enum CBLAS_TRANSPOSE TransB,
4     blasint m, blasint n, blasint k,
5 #ifndef COMPLEX
6     FLOAT alpha,
7     IFLOAT *a, blasint lda,
8     IFLOAT *b, blasint ldb,
```

```
9       FLOAT beta,
10      FLOAT *c, blasint ldc) {
11  #else
12      void *valpha,
13      void *va, blasint lda,
14      void *vb, blasint ldb,
15      void *vbeta,
16      void *vc, blasint ldc) {
17  ...
18  #endif
```

Listing 3.3: OpenBLAS GEMM interface.

3.3.3 Hardware Configuration

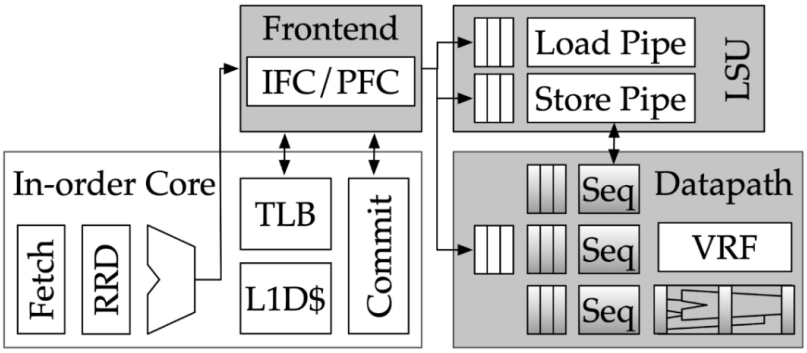

Figure 3.8: Saturn vector unit.

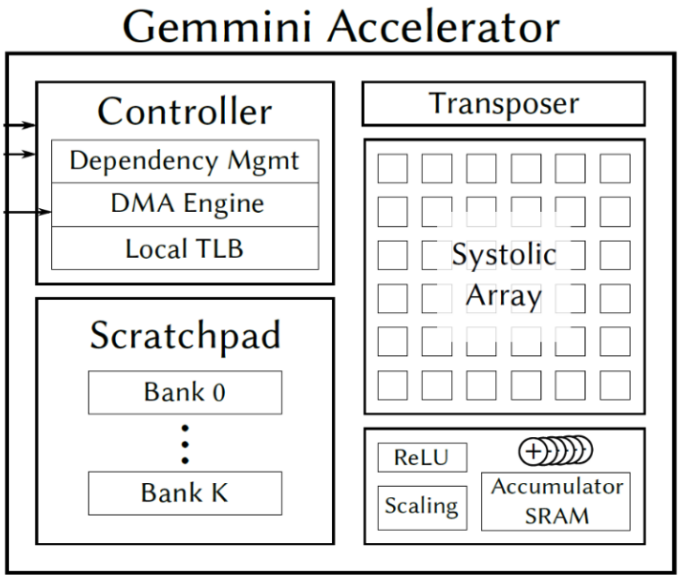

Figure 3.9: Gemmini accelerator.

Both the vector backend and the Gemmini backend rely on specialized hardware accelerators within the SoC to function. For the vector backend, the Saturn vector unit with a 256-bit vector register width, developed in the Chipyard framework, is used to execute RVV instructions. For the Gemmini backend, a Gemmini accelerator configured with support for single-precision floating-point data is used. Both hardware units are integrated into the same SoC alongside a dual-issue Shuttle core. Figure 3.8 shows the architecture of the Saturn vector unit, and Figure 3.9 shows the architecture of the gemmini accelerator.

Two SoCs are configured, as shown in code block 3.4 and code block 3.5. The first configuration consists of a Shuttle core integrated with a Saturn vector unit, while the second includes both the Saturn vector unit and a Gemmini accelerator, alongside the same Shuttle core. The first SoC is used to benchmark the vector backend in isolation, without invoking any Gemmini-specific instructions. In contrast, the second SoC is used to evaluate the Gemmini backend. The first configuration is preferred for benchmarking the vector backend due to its significantly higher simulation speed, enabled by its simpler hardware design. SoC simulations are conducted using FireSim on the Xilinx Alveo U250 FPGA platform. In simulation, the first SoC operates at 100 MHz, whereas the second SoC operates at only 7 MHz, but this only impacts the amount of time finishing the simulation. Notably, even when evaluating the Gemmini backend, the presence of the Saturn vector unit remains necessary, as the compiler is configured with vector instructions enabled.

```
class GENV256D128FPGemminiShuttleConfig extends Config(
    new gemmini.ChipFP32GemminiConfig ++
    new saturn.shuttle.WithShuttleVectorUnit(256, 128, VectorParams.genParams
    ) ++
    new chipyard.config.WithSystemBusWidth(128) ++
    new shuttle.common.WithShuttleTileBeatBytes(16) ++
    new shuttle.common.WithNShuttleCores(1) ++
    new chipyard.config.AbstractConfig
)
```

Listing 3.4: SoC configuration of one Saturn vector unit and one Shuttle core.

```
class GENV256D128ShuttleConfig extends Config(
    new saturn.shuttle.WithShuttleVectorUnit(256, 128, VectorParams.genParams
    ) ++
    new chipyard.config.WithSystemBusWidth(128) ++
    new shuttle.common.WithShuttleTileBeatBytes(16) ++
    new shuttle.common.WithNShuttleCores(1) ++
    new chipyard.config.AbstractConfig
)
```

Listing 3.5: SoC configuration of one Saturn vector, one Gemmini accelerator, and one Shuttle core.

Chapter 4

Results and Discussion

4.1 Video Encoding

The video encoding methods are independently evaluated using a sample clip from *Big Buck Bunny*, a standard dataset widely used in video compression research. While the system supports stereo image encoding and decoding by applying the algorithm to each image separately, the evaluation here is performed on a single image to ensure visual clarity and consistency with conventional video datasets. Figure 4.1 presents an image of the original, perceptually lossless video. Figure 4.2 shows a frame encoded using the H.264 codec with a Constant Rate Factor of 28. As expected, compression artifacts are visible due to the lossy nature of H.264. Figure 4.3 displays a frame encoded with foveated video encoding applied on top of H.264. For this configuration, the Gaussian standard deviation σ is set to 0.2, and the quantization offset strength δ is set to 30 to deliberately blur the peripheral regions for visual demonstration. As shown, the central region on Figure 4.3 retains higher visual fidelity than the standard H.264-encoded image, while the peripheral areas are intentionally more blurred to emphasize the effect of foveated encoding.

4.2 Compute Offloaded Systems

The compute offload system is benchmarked in conjunction with foveated video encoding, as evaluating the compute offload mechanism in isolation would require transferring full-resolution RGBA images, which imposes impractically high bandwidth requirements. Furthermore, since both the graphics compute offload and foveated video encoding are newly developed features, the ILLIXR system is executed on the host machine rather than within the FireSim simulation environment. This ensures functional correctness and allows performance evaluation in a native setting consistent with the original ILLIXR deployment. The platform on which the system is evaluated is a 12th Gen Intel Core i9-12900K CPU with an NVIDIA RTX 4070 Ti GPU. For the application workload, the open-source renderer Filament is used. As part of this work, OpenXR support was added to the Filament renderer, enabling it to serve as a representative VR application. The renderer supports the rendering of models in the GLTF format, providing flexibility for evaluation scenarios.

The complete system consists of three processes running on the host machine: the Filament renderer, the Monado runtime, and the ILLIXR system. The OpenXR application,

Figure 4.1: An image from the perceptually lossless sample video.

Figure 4.2: An image from the sample video encoded by H.264.

Figure 4.3: An image from the sample video encoded by H.264 with foveated video encoding applied.

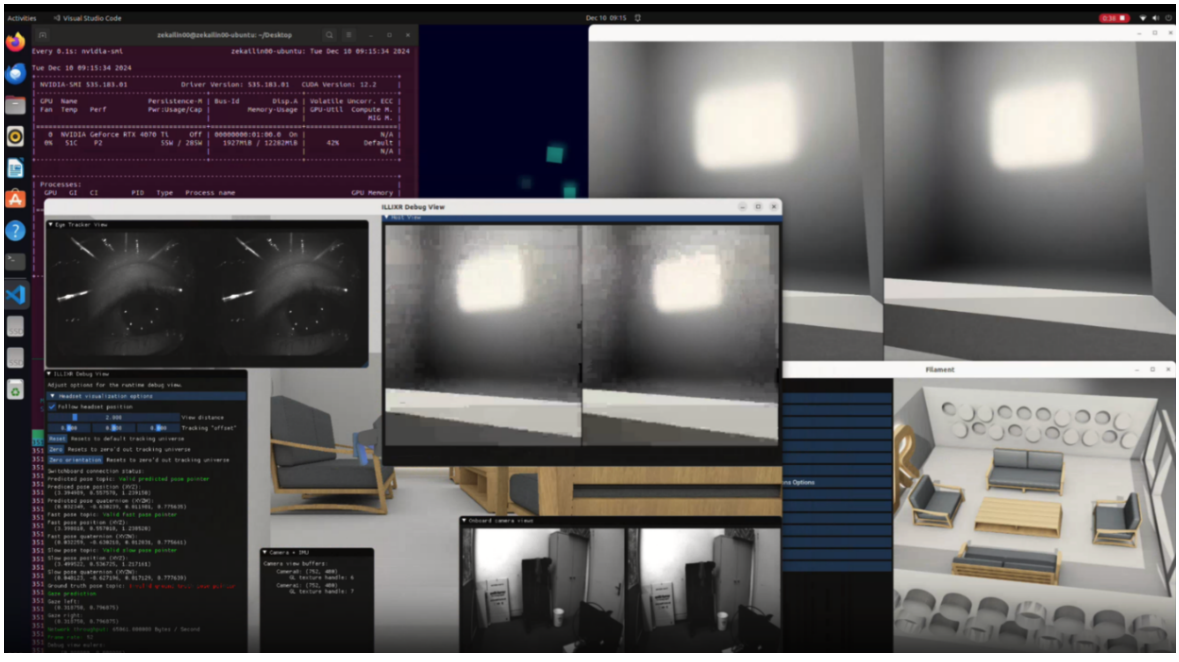

Figure 4.4: Graphics compute offload system.

implemented using the Filament renderer, generates three images every 20 milliseconds, two for stereo rendering and one for a 2D preview window, limited by synchronization with the OpenXR runtime through `xrWaitFrame`. The Monado runtime applies a timewarp, composes the stereo views into a single image, encodes the resulting frame, and transmits the encoded bitstream to ILLIXR approximately every 23 milliseconds. The majority of Monado's processing time is spent on video encoding (approximately 7.5 ms) and image composition (approximately 13 ms). On the ILLIXR side, each system component runs in a dedicated thread with a continuous execution loop. Gaze inference takes approximately 480 milliseconds per iteration, while video decoding requires about 23 milliseconds on average. For processing offline camera data, the ILLIXR plugin reads image frames from disk and publishes them to consumer plugins at intervals of 50 milliseconds.

Figure 4.4 demonstrates a graphics compute offload system. Given an image resolution of 1280×720 pixels, a Constant Rate Factor of 28, a quantization offset strength of 30, and a Gaussian function standard deviation of 0.2, the resulting bitrate of the socket connection varies between 48 kbps and 150 kbps depending on the visual complexity of the rendered content. Overall, the system operates at approximately 50 frames per second, with gaze data updates occurring at a frequency of 2 Hz.

The primary performance bottlenecks in the current rendering pipeline occur during the image encoding and decoding stages. Prior to encoding, image data must be transferred from GPU memory to system memory in a synchronized manner. This requires a Vulkan fence at the end of the Monado compositor, which introduces significant delays in the graphics pipeline. Furthermore, both the encoder and decoder currently operate on the CPU, requiring memory copies to move the x264 bitstream into, and decoded images out of, the codec context. These additional memory transfers further degrade system throughput. In the case of eye tracking, although the neural network has been offloaded to a GPU, the corresponding kernel has not yet been optimized for parallel execution, resulting in a limited inference rate of only two inferences per second.

A potential architectural improvement involves relocating the timewarp operation. At present, a timewarp is performed within the compositor of the Monado runtime. However, executing the timewarp on the ILLIXR side, immediately before the final image is displayed, could help mitigate the negative impact of network latency. This design shift would also align the final pose correction more closely with the actual moment of image presentation, improving visual stability and reducing perceived motion lag.

4.3 Visual Inertial Odometry

The OpenVINS binary is uploaded to an Ubuntu 24 image, along with a dataset comprising stereo camera recordings and inertial measurements from an accelerometer and gyroscope. The visual-inertial odometry workload is evaluated using the dataset that comes with the ILLIXR system. In the initial 94 frames, only the frontend module is active due to the insufficient number of tracked feature points required to trigger backend processing. Between frames 94 and 135, only the MSCKF algorithm is executed; however, the feature points are not yet persistent enough to be promoted to SLAM features. Starting from frame 136, longer-term feature points emerge, enabling the execution of all components in the OpenVINS library.

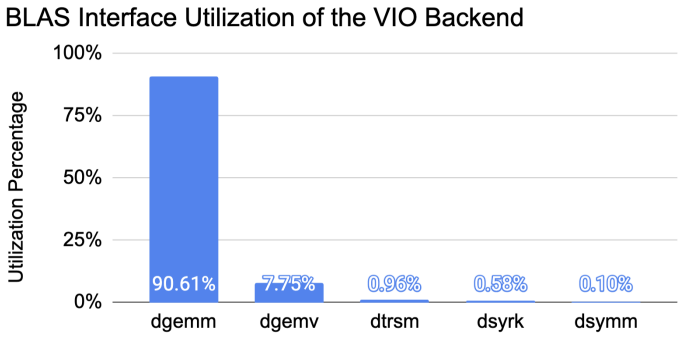

Figure 4.5: BLAS interface utilization of the VIO backend.

Initial benchmarking was conducted on OpenVINS running on the Chipyard Shuttle core integrated with Saturn vector units. The frequency of BLAS API calls generated by OpenVINS was observed to vary over time, depending on the number of visual features returned by the frontend. In later frames, when a sufficient number of features are available for accurate pose estimation, GEMM becomes the dominant operation, accounting for approximately 90 percent of all BLAS calls. The next most frequent operation is GEMV, comprising roughly 8 percent of the calls. Figure 4.5 shows the utilization of the BLAS interface by the VIO backend.

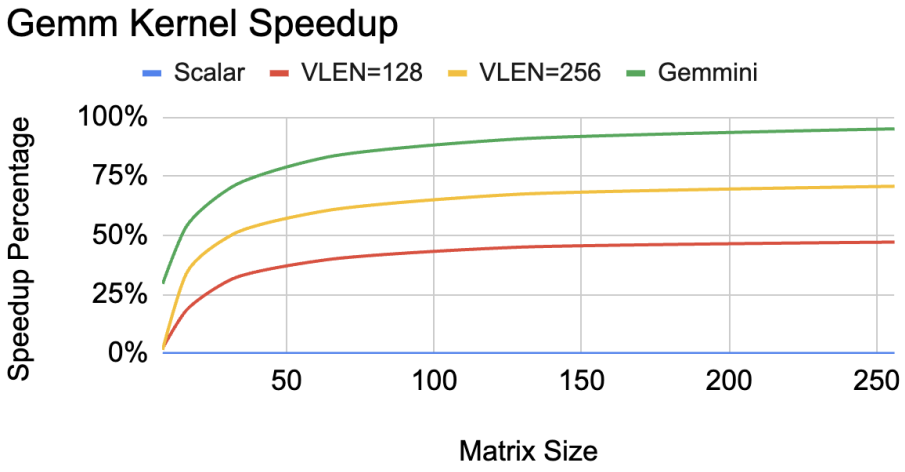

Figure 4.6: GEMM kernel speedup relative to matrix dimensions.

To assess performance at the kernel level, OpenBLAS was benchmarked using various RISC-V backend implementations. Specifically, the GEMM kernel was evaluated across four configurations: a scalar implementation, a vectorized kernel targeting 128-bit vector registers, a kernel optimized for 256-bit vector registers, and a Gemmini kernel. The speedup is shown in Figure 4.6. As the matrix size increases, the observed speedup converges. Compared to the scalar implementation, the kernel configured for `vlen=128` achieves a 50% speedup, corresponding to a 2× reduction in cycle count. The `vlen=256` kernel yields a 70% speedup, or

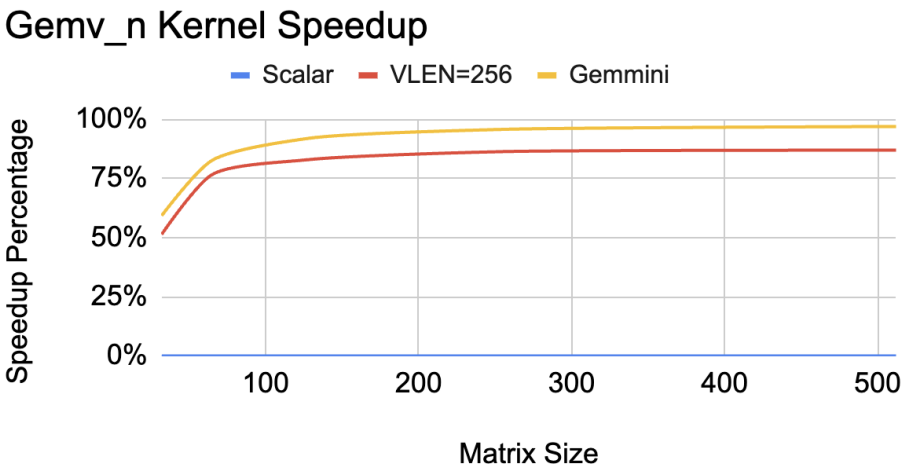

Figure 4.7: GEMV_N kernel speedup relative to matrix dimensions.

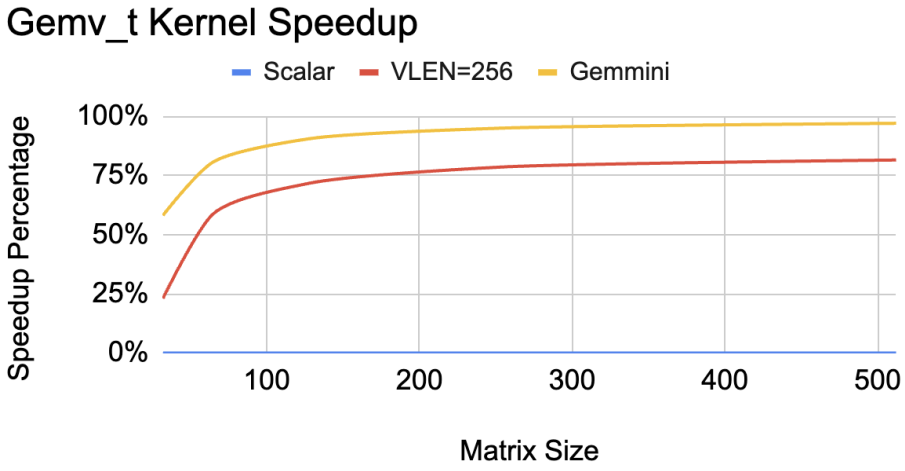

Figure 4.8: GEMV_T kernel speedup relative to matrix dimensions.

a 3.3× reduction in cycle count. The Gemmini kernel demonstrates the highest performance improvement, achieving a 95% speedup, equivalent to a 20× reduction in cycle count. For GEMV, two variants, `GEMV_T` and `GEMV_N`, were evaluated, both demonstrating comparable performance improvements over the scalar baseline. Figure 4.7 and Figure 4.8 show the GEMV kernel speedup relative to matrix dimensions. The `vlen=256` kernel converges to an 85% speedup, corresponding to a 6.6× reduction in cycle count, while the Gemmini kernel reaches a 97% speedup, equivalent to a 33.3× reduction.

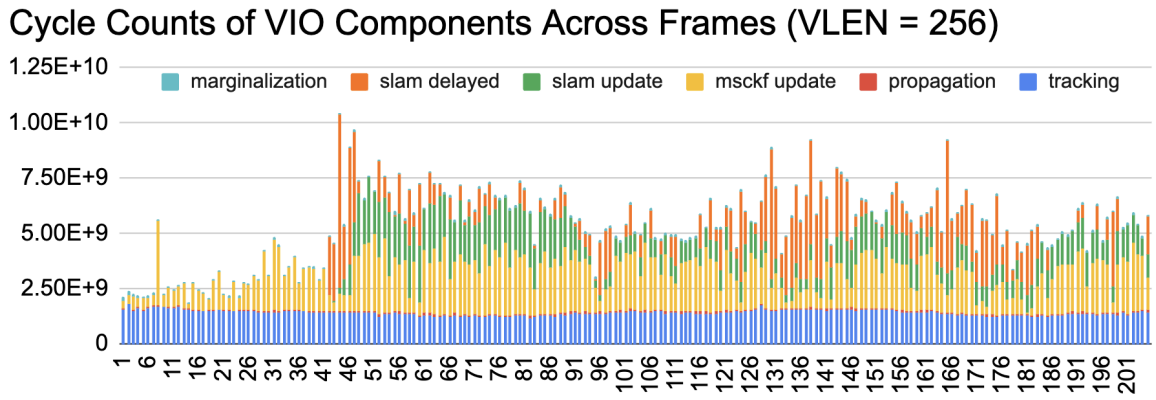

Figure 4.9: Cycle counts of VIO components across the frames on the Saturn backend (vlen=256).

For application-level benchmarking, Figure 4.9 presents the cycle counts of various components within the OpenVINS library across successive frames. The data is collected starting from the point at which the backend is activated with a sufficient number of tracked features, and the evaluation is performed using the `vlen=256` backend. The tracking component corresponds to the frontend, which employs OpenCV to process incoming images and extract feature points. As shown in the plot, its cycle count remains relatively stable across frames, indicating a consistent computational workload. The propagation component represents the prediction stage of the VIO backend, where inertial measurements from the IMU are integrated to estimate the device's pose. The MSCKF update component corresponds to the correction stage and displays variation in cycle count depending on the number of visual features propagated from the frontend that require processing. The marginalization step, which discards obsolete or redundant features to maintain computational efficiency, incurs a relatively low cost and contributes minimally to the overall cycle count. In addition to the core VIO pipeline, OpenVINS includes support for managing long-term features used in SLAM. The SLAM delayed and SLAM update components are responsible for processing and updating these features.

Figure 4.10 illustrates the performance improvement achieved by utilizing the vector backend with a vector length of 256 bits, in comparison to the scalar backend. The observed speedup varies significantly across frames. This variation is likely attributed to floating-point rounding limitations in other BLAS interfaces—such as `TRSM`, `SYRK`, and `SYMM`—that internally invoke the same GEMM kernels. Since the vector backend is implemented at the kernel level, it affects not only `GEMM` operations but also those of the aforementioned interfaces, which are also used in OpenVINS. These numerical discrepancies can alter feature processing results,

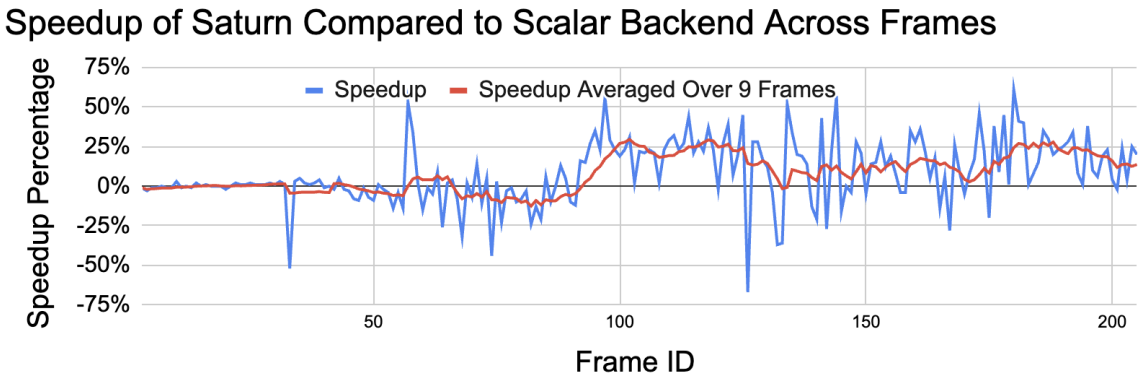

Figure 4.10: Speedup of Saturn backend (vlen=256) compared to scalar backend across the frames.

leading to different behaviors in promoting or discarding features as SLAM candidates over time.

The average application-level speedup over 200 frames is approximately 8.49%. When a moving average with a window size of 9 frames is applied, the speedup initially starts near 0% but gradually increases, reaching approximately 25% in later frames. Despite the kernel-level speedup reaching approximately 70% with the vector backend, the application-level improvement remains limited. This disparity underscores the importance of hardware–software co-design to translate low-level computational gains into meaningful system-level performance improvements for visual-inertial odometry workloads.

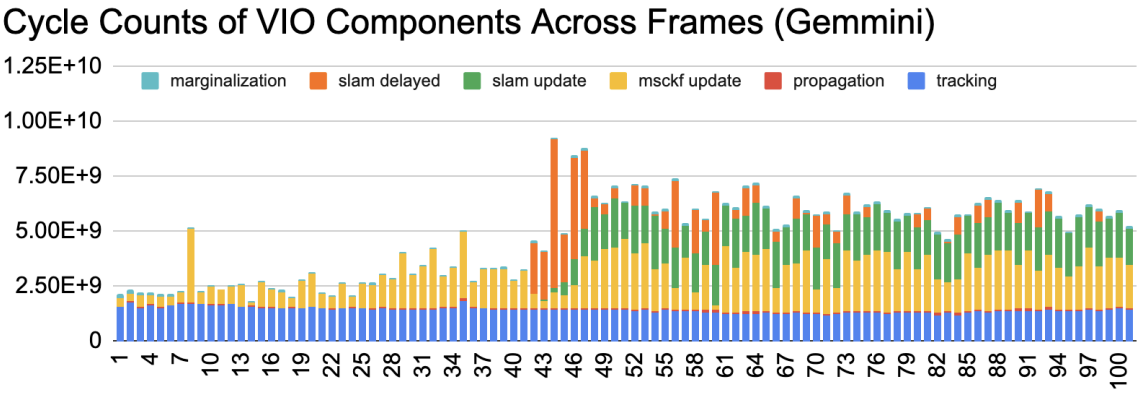

Figure 4.11: Cycle counts of VIO components across the frames on the Gemmini backend.

Figure 4.11 presents the cycle counts of various visual-inertial odometry (VIO) components across successive frames when executed on the Gemmini backend. As with the vector backend, frames are recorded beginning from the point at which the frontend consistently returns a sufficient number of tracked features to activate backend processing. The overall distribution and temporal pattern of cycle counts closely resemble those observed on the vector backend, indicating similar computational behavior across both hardware configurations.

Figure 4.12 illustrates the speedup achieved by the Gemmini backend across frames. In contrast to the results obtained with the vector backend, the speedup observed with the Gemmini backend exhibits minimal variation and remains relatively consistent over time, with an average speedup of approximately 7.7%. This consistency is primarily due to differences in implementation: the Gemmini backend is integrated at the `GEMM` interface level within the OpenBLAS library, whereas the vector backend is implemented at the kernel level. Consequently, only `GEMM` operations benefit from the Gemmini backend, while other Level 3 BLAS routines—such as `TRSM`, `SYRK`, and `SYMM`—continue to rely on the default scalar backend. This limited scope of acceleration results in lower variability in performance across frames.

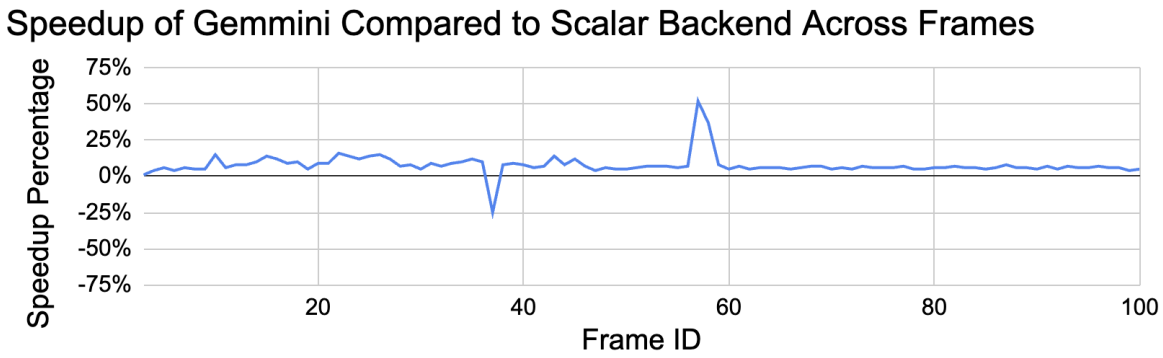

Figure 4.12: Speedup of Gemmini backend (vlen=256) compared to scalar backend across the frames.

Chapter 5

Conclusions

This work presents a comprehensive exploration of software–hardware co-design for virtual reality systems, targeting two of the most computationally demanding components: rendering and localization. By integrating the ILLIXR software framework with Berkeley's Chipyard and FireSim hardware infrastructure, this research enables full-stack evaluation and optimization across both domains. A graphics compute offload system was implemented at a level below the OpenXR abstraction, allowing any compliant application to benefit from offloading rendering tasks to a host machine. To mitigate the resulting bandwidth challenges, a foveated video encoding pipeline was developed, leveraging gaze prediction to reduce bitrate while preserving visual quality. On the localization side, the visual-inertial odometry workload was accelerated using both RISC-V vector intrinsics and the Gemmini hardware accelerator. Experimental results demonstrated significant kernel-level speedups and identified application-level bottlenecks, highlighting the importance of holistic system design. Overall, this study provides a concrete foundation for future XR platform development, where domain-specific hardware acceleration and cross-layer optimization are critical to balancing performance, efficiency, and usability.

Chapter 6

Future Work

This work represents an initial step toward the full-stack design and evaluation of compute offloaded virtual reality systems, bridging both software and hardware domains. It addresses a critical need in the future of XR: enabling compact, lightweight headsets with high computational capability suitable for everyday use. While the current implementation lays the foundation, substantial work remains. As a next step, the ILLIXR system needs to be benchmarked within FireSim to assess the performance impact of the newly integrated compute offload components. On the visual-inertial odometry side, while the Gemmini backend provides acceleration for GEMM and GEMV operations, additional support for the full BLAS interface is required. Beyond refining the current system, further research can explore how different hardware accelerators or software architectures affect overall XR system performance. For instance, the VIO frontend workload could be offloaded to a GPU, or the eye-tracking module could be moved from the headset to the host system to evaluate the impact on latency and efficiency. This work establishes a foundation for systematically exploring the XR system design space and co-optimizing hardware and software for next-generation immersive experiences.

References

Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanović, K. and Nikolić, B. (2020), 'Chipyard: Integrated design, simulation, and implementation framework for custom socs', *IEEE Micro* **40**(4), 10–21.

Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H. and Waterman, A. (2016), The rocket chip generator, Technical Report UCB/EECS-2016-17.
URL: *http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html*

Calonder, M., Lepetit, V., Strecha, C. and Fua, P. (2010), Brief: Binary robust independent elementary features, Vol. 6314, pp. 778–792.

Chaudhary, A. K., Kothari, R., Acharya, M., Dangi, S., Nair, N., Bailey, R., Kanan, C., Diaz, G. and Pelz, J. B. (2019), Ritnet: Real-time semantic segmentation of the eye for gaze tracking, *in* '2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)', IEEE, p. 3698–3702.
URL: *http://dx.doi.org/10.1109/ICCVW.2019.00568*

Genc, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P., Zhao, J., Grubb, D., Liew, H., Mao, H., Ou, A., Schmidt, C., Steffl, S., Wright, J., Stoica, I., Ragan-Kelley, J., Asanovic, K., Nikolic, B. and Shao, Y. S. (2021), 'Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration'.
URL: *https://arxiv.org/abs/1911.09925*

Geneva, P., Eckenhoff, K., Lee, W., Yang, Y. and Huang, G. (2020), Openvins: A research platform for visual-inertial estimation, *in* '2020 IEEE International Conference on Robotics and Automation (ICRA)', pp. 4666–4672.

Goto, K. and Geijn, R. A. v. d. (2008), 'Anatomy of high-performance matrix multiplication', *ACM Trans. Math. Softw.* **34**(3).
URL: *https://doi.org/10.1145/1356052.1356053*

Harris, C. G. and Stephens, M. J. (1988), A combined corner and edge detector, *in* 'Alvey Vision Conference'.
URL: *https://api.semanticscholar.org/CorpusID:1694378*

Huzaifa, M., Desai, R., Grayson, S., Jiang, X., Jing, Y., Lee, J., Lu, F., Pang, Y., Ravichandran, J., Sinclair, F., Tian, B., Yuan, H., Zhang, J. and Adve, S. V. (2022), 'Illixr: An open testbed to enable extended reality systems research', *IEEE Micro* **42**(4), 97–106.

Illahi, G. K., Siekkinen, M., Kämäräinen, T. and Ylä-Jääski, A. (2020), On the interplay of foveated rendering and video encoding, *in* 'Proceedings of the 26th ACM Symposium on Virtual Reality Software and Technology', VRST '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3385956.3422126*

Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R., Bachrach, J. and Asanovic, K. (2018), Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud, *in* '2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)', pp. 29–42.

Mourikis, A. I. and Roumeliotis, S. I. (2007), A multi-state constraint kalman filter for vision-aided inertial navigation, *in* 'Proceedings 2007 IEEE International Conference on Robotics and Automation', pp. 3565–3572.

Nikiforov, D., Dong, S. C., Zhang, C. L., Kim, S., Nikolic, B. and Shao, Y. S. (2023), RosÉ: A hardware-software co-simulation infrastructure enabling pre-silicon full-stack robotics soc evaluation, *in* 'Proceedings of the 50th Annual International Symposium on Computer Architecture', ISCA '23, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3579371.3589099*

Rosten, E. and Drummond, T. (2005), Fusing points and lines for high performance tracking, *in* 'Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1', Vol. 2, pp. 1508–1515 Vol. 2.

Saxena, A., Chiu, C.-Y., Menke, J., Shrivastava, R. and Sastry, S. (2022), 'Simultaneous localization and mapping: Through the lens of nonlinear optimization'.
URL: *https://arxiv.org/abs/2112.05921*

Wang, Q., Zhang, X., Zhang, Y. and Yi, Q. (2013), Augem: automatically generate high performance dense linear algebra kernels on x86 cpus, *in* 'Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis', SC '13, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/2503210.2503219*

Wiedemann, O., Hosu, V., Lin, H. and Saupe, D. (2020), Foveated video coding for real-time streaming applications, *in* '2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)', pp. 1–6.

Zhao, J., Grubb, D., Rusch, M., Wei, T., Anderson, K., Nikolic, B. and Asanović, K. (2024), The saturn microarchitecture manual, Technical Report UCB/EECS-2024-215.
URL: *http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-215.html*

Zhao, J., Korpan, B., Gonzalez, A. and Asanovic, K. (2020), 'Sonicboom: The 3rd generation berkeley out-of-order machine'.