OPEN ACCESS

Research Article

Performance Characterization of gr-owc, the GNURadio Out-of-Tree Module for Optical Wireless Communications

Kunal P. Sangurmath,[1] Michael B. Rahaim ✉,[2]

[1] Computer Science Department, University of Massachusetts, Boston, MA, USA;
[2] Engineering Department, University of Massachusetts, Boston, MA, USA;

Article History

Submitted: January XX, XXXX Accepted: January XX, XXXX Published: February XX, XXXX

Abstract

As researchers continue to explore novel technologies for future generations of wireless communications, flexible and adaptable tools are required for testing and development. Software-Defined Radios (SDRs) allow for flexible and reconfigurable signal processing, making them a crucial tool for rapid testing and experimental validation of communication systems. GNURadio is an open source SDR toolkit with functionality to extend it's core software library by developing Out-Of-Tree (OOT) modules. Our gr-owc OOT module enables SDR capabilities for Optical Wireless Communication (OWC) and serves as a research tool that facilitates real-time experimentation, system-level analysis, and algorithm development in OWC. To effectively support real-time experimentation, the custom signal processing blocks in gr-owc must be computationally efficient to keep up with real-time hardware requirements. This paper presents a study of the Central Processing Unit (CPU) utilization patterns of gr-owc's signal processing blocks. Namely, we provide insights into the computational efficiency of gr-owc blocks and sample flowgraphs by analyzing CPU utilization with respect to sample rate, hardware deployments, and block configurations. We also introduce recent modifications to these blocks that improve performance while maintaining functionality. Experimental results validate the influence of parameters like compute hardware and coding implementation on CPU usage and reveal the limitations imposed by single-core block processing in GNURadio applications.

Keywords:

Optical Wireless Communication (OWC), Software Defined Radio (SDR), Visible Light Communication (VLC), GNURadio, CPU utilization, Real-time signal processing.

1. Introduction

Optical wireless communication (OWC) has gained significant interest from the research community in recent years, initially focusing on point-to-point links and novel modulation techniques. Recently, research has shifted towards higher-layer design for multi-cell and multi-user systems, emphasizing resource allocation and mobility management. Users in multi-cell environments or within a cell shared by multiple users face additional challenges such as dynamic resource allocation, handover management, and interference mitigation [1,2]. Novel algorithms have been introduced and evaluated in theory and in simulation [3–5], but experimental validation of techniques for multi-node OWC systems remains limited. This is primarily due to the complexity of developing proof-of-concept implementations that accurately model interference, mobility, and real-world deployment constraints while also providing control at the signal level. To address this, we leverage software-defined radio (SDR) tools and apply SDR techniques for OWC research. In particular, we use the GNURadio SDR toolkit with custom software and hardware modifications for OWC signal processing.

*** Corresponding Author:**
Michael B. Rahaim,
Engineering Department, University of Massachusetts, Boston, MA, USA,
Michael.Rahaim@umb.edu; Tel.: +1-617-287-4132

GNURadio is a widely used SDR toolkit, valued for its accessibility and flexibility in RF research and experimental validation [6]. While GNURadio has a vast library of signal processing tools, it also allows for development of custom Out-Of-Tree (OOT) modules that extend it's core functionality. Previously, we have introduced the *gr-owc* OOT module for OWC [7–10]. It provides a suite of custom signal processing blocks, documentation, and sample flowgraphs that facilitate real-time experimentation, system-level analysis, and algorithm development in multi-node OWC networks. The signal processing blocks and sample flowgraphs serve as a "golden reference" for evaluating novel techniques and improvements, supporting further innovation in the field.

Key advantages of SDR include it's ability to support simulation-to-experimentation workflows and it's configuration flexibility. First, it enables researchers to validate their techniques in a simulated environment before deploying them in experimental setups with minimal adjustments. Second, its flexibility allows for the interchange of front-end transmitter (Tx) and receiver (Rx) configurations while maintaining common signal processing chains and test flowgraphs, ensuring consistency across different experimental setups. The gr-owc module brings the benefits of SDR to the OWC research field and bridges the gap between simulation and real-world experimentation, offering a cost-effective and flexible solution compared to traditional OWC hardware, which is often expensive and inflexible. By leveraging gr-owc and the broader GNURadio codebase, researchers can validate new OWC techniques in a simulated environment and seamlessly transition to experimental setups without significant modifications.

Compared to offline simulation tools, real-time SDR-based test systems allow for experimental setups that emulate practical scenarios and validate system designs under realistic constraints. However, they also require significant attention to potential processing bottlenecks. This is particularly true when the signal processing is implemented on a general purpose processor. SDR tools like GNURadio offer a robust framework to implement and test signal processing methods in both simulated and experimental environments. However, the performance limits of individual blocks should be understood before moving to experimental systems in order to identify bottlenecks. Identifying these bottlenecks allows us to determine where targeted improvements are needed, potentially by FPGA offloading (e.g., with GNURadio's RFNoC tools).

In this work, we test gr-owc's processing blocks to evaluate their ability to meet real-time processing requirements. Our evaluation focuses on assessing the performance consistency of gr-owc blocks across different compute hardware configurations and software implementations. We also demonstrate the performance improvements from recent code modifications for various blocks within the gr-owc module. While our prior work in [7–10] demonstrates baseline functionality, this work highlights modifications that improve the real-time processing performance. Furthermore, the introduced code modifications and corresponding comparative analysis also offer valuable insights for other researchers developing custom blocks and/or OOT modules in GNURadio.

The rest of this paper is organized as follows: Section 2 provides background on SDR, GNURadio, and the gr-owc module. Section 3 describes the experimental test procedure and methodology used for CPU utilization measurements. Section 4 presents our testing configurations along with an evaluation of the results. Section 5 summarizes key findings.

2. Background

SDR has become a key enabler of configurable RF communication systems. Current work in SDR often highlights the functional ability to implement dynamic and adaptive systems. However, the ability to quickly move ideas from theory or simulation to practice is another significant aspect of the "software-defined" concept. This design flexibility has been incredibly valuable to the research community. By reducing the barrier to entry and making it more feasible to physically instantiate novel ideas, SDR has created a more equitable opportunity for experimental research in the field of wireless communications.

2.1. The GNURadio SDR Toolkit

SDRs provide a flexible platform for experimental validation of communication systems, and GNURadio is one of the most widely used toolkits for developing SDRs. In GNURadio, various signal processing *blocks* can be connected within a *flowgraph* to create custom signal processing chains. GNURadio also offers functionality to create custom blocks and OOT modules. These OOT modules exist outside the GNURadio source tree. Developers create OOT modules to extend GNURadio with custom functions and signal processing blocks while maintaining their own independent codebase, and other users can install OOT modules as desired. This approach allows for development of additional functionality alongside the main GNURadio software, and sharing of the functionality with the broader research community.

2.2. The gr-owc OOT Module

The gr-owc OOT module [7] extends GNURadio's capabilities to support OWC techniques, offering custom signal processing blocks and flowgraphs for OWC research. It provides a set of parameterized signal processing blocks that enable researchers to implement, simulate, and test OWC systems. The module includes OWC-specific signal processing blocks, comprehensive documentation, and sample flowgraphs that serve as reference implementations for various OWC applications. Specific gr-owc blocks support various modulation and demodulation techniques, along with a configurable OWC channel model. The seamless integration with GNURadio offers flexibility for diverse flowgraphs that combine gr-owc blocks with signal processing blocks from the core GNURadio library. Additionally, it supports real-time signal processing and introduces hardware considerations for applying SDR concepts to OWC, ultimately facilitating the transition from simulation to experimental validation [8–10].

3. Materials and Methods

In order to evaluate real-time performance of the blocks within gr-owc, we implemented a common flowgraph structure to isolate block performance and observe CPU utilization as a function of sample rate. A set of relevant blocks from gr-owc have been tested across different hardware platforms and with different code structure in order to demonstrate the impacts on real-time performance and highlight potential bottlenecks to be addressed.

3.1. Test Procedure

Our testing setup for gr-owc blocks follows the standardized flowgraphs in Fig. 1 to isolate the performance of a given block as our unit under test (UUT). In multi-rate flowgraphs (i.e., flowgraphs where the UUT acts as an interpolator or decimator), the location of the throttle impacts the relative relationship between the throttle's sample rate and CPU utilization. Therefore, we consider two versions of the test flowgraph. With the OOK modulator, flowgraph V1 evaluates performance relative to bit rate whereas V2 evaluates performance relative to sample rate.

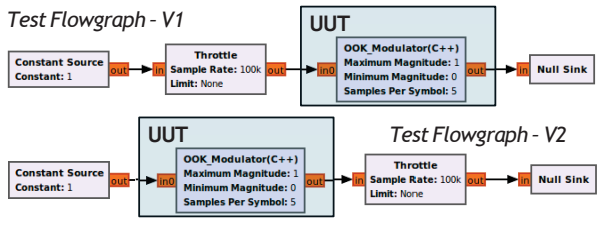

Figure 1: Example performance testing flowgraphs with gr-owc's On-Off-Keying Modulator block as the Unit Under Test (UUT).

These flowgraphs serve as the baseline for all evaluations. Besides the UUT, the remaining blocks consume minimal CPU resources as they perform lightweight operations and ensure that the UUT performance is isolated without adding significant overhead.

- **Constant Source**: Generates a stream of samples without performing computation-heavy processes.
- **Throttle**: Copies items from input to output while enforcing an upper limit on the sample rate. It does not modify the signal, so CPU usage is minimal.
- **Null Sink**: Discards the incoming data without processing or storage, requiring low CPU resources.

To analyze CPU utilization trends, our test systematically sweeps through different sample rate values[1], measuring CPU usage in each instance. For each sample rate, CPU utilization is recorded using *pidstat* tool by monitoring the Process ID of the GNURadio process. A total of 15 CPU readings were collected per sample rate, and the results were averaged to obtain a stable measurement. GNURadio uses a thread-per-block scheduler [11], where each block runs in an individual thread that cannot utilize more than one CPU core, which limits each block to single-core execution. However, processing for different blocks in a flowgraph can be distributed among cores.

The Throttle block regulates the overall flow of samples by setting an upper limit on the sample rate. However, it does not enforce a minimum sample rate. If downstream blocks cannot keep up with the specified rate, the flow of samples is reduced throughout the flowgraph. Therefore, when the CPU utilization saturates (i.e., stops increasing with respect to the throttle's sample rate parameter), this indicates that the UUT's processing capacity, rather than the Throttle block's specified sample rate, is the primary bottleneck. We define this as the peak sample rate for a given block in a given test scenario.

Knowledge of the peak sample rate is critical in determining whether the UUT can process incoming samples at the specified rate or if CPU constraints prevent it from keeping up, leading to processing delays or dropped samples. This analysis helps assess the real-time processing capabilities of gr-owc blocks and their ability to scale with increasing sample rates. It also offers insight into the potential for a given flowgraph to keep up with desired

[1]Here, sample rate refers to the setting of the Throttle block irrespective of whether samples flowing through the Throttle are true samples, bits, or other data.

Table 1: System hardware configurations.

System	Processor	Physical Core(s)	Thread(s) / Core
System 1	Intel Core i5 650 (3.20 GHz)	**2**	**2**
System 2	Intel Xeon Gold 6230 (2.10 GHz)	**20**	**1**
System 3	Raspberry Pi 5 (2.40 GHz)	**4**	**1**

hardware sample rates when moving to experimental test cases where sample rate is controlled by SDR hardware rather than a throttle block.

3.2. Compute Hardware

The testing of gr-owc processing blocks was conducted on three different compute systems (Table 1) in order to evaluate how performance is impacted by characteristics of the compute hardware [12]. Since a major focus of gr-owc is to enable low-cost proof-of-concept experimentation for OWC systems, we demonstrate performance on a range of hardware from a Raspberry Pi microcontroller to a higher performance multi-core system. For all test systems, the Ubuntu 22.04.3 LTS operating system was installed.

The three systems differ mainly in how physical cores and logical threads affect instruction-level performance. System 1 (Intel Core i5-650) supports two logical threads per core, so both threads share the same core resources, which can reduce the effective IPC (Instructions Per Cycle) for each thread when they run at the same time. System 2 (Intel Xeon Gold 6230) provides many physical cores with one thread per core, allowing each thread to run independently and achieve higher and more stable IPC. System 3 (Raspberry Pi 5) also uses one thread per core, but with fewer cores and a lower clock frequency. In general, systems with more physical cores and a single thread per core deliver more consistent IPC per thread, while systems with multiple threads per core trade per-thread IPC for higher overall utilization.

3.3. Code Implementations

Signal processing blocks within OOT modules can be implemented in Python, C++, or C++ with VOLK. Each approach provides distinct benefits in terms of development complexity and real-time performance, allowing developers to select the approach that aligns with their performance and development needs.

- **Python**: Ideal for non-time-critical tasks due to its ease of use, faster development cycles, and flexibility. However, Python implementations are slower and less desirable for high-performance real-time applications [13].
- **C++**: Offers significantly better performance, making it suitable for real-time signal processing. It provides low-level control and faster execution but requires more effort for development and debugging.
- **C++ with VOLK**: The Vector-Optimized Library of Kernels (VOLK) is an open-source library that provides a collection of SIMD (Single Instruction, Multiple Data) kernels designed to accelerate mathematical operations. It enhances C++ implementations by leveraging SIMD instructions for repetitive computationally intensive tasks.

Prior publications related to gr-owc focused primarily on functionality (i.e., blocks developed in Python). In this work, we demonstrate the real-time performance improvements associated with our recent development of C++ blocks for components in the gr-owc module. This work is also based on the gr-owc codebase which has been modified for compatibility with GNURadio Release 3.10, whereas our prior publications considered an earlier version of gr-owc which was compatible with Release 3.8.

4. Results and Discussion

When integrating GNU Radio SDR flowgraphs with hardware platforms such as USRP, throughput and latency limitations can arise from three primary sources. First, different USRP models support different maximum throughput capacities, which can introduce a bottleneck [14] if higher sample (data) rates are transmitted to the hardware. Second, the communication interface between the host system and the USRP, such as Ethernet, imposes bandwidth limitations in terms of achievable bits per second and associated transmission latency. Third, based on the design and implementation of the GNU Radio SDR blocks, the host system processor must be capable of meeting the real-time processing requirements imposed by the front-end hardware.

In this work, the focus is on the third constraint, namely the processing capability of the host system executing the GNU Radio flowgraph. The evaluation characterizes whether the system processor can sustain real-time operation under increasing sample rates. This allows

isolation of the computational performance of the gr-owc blocks with respect to real-time feasibility.

To provide detailed analysis of specific blocks within gr-owc, we focus on a subset of blocks. Namely, we analyze the OWC **channel modeling blocks** (considered for real-time simulation alongside experimental deployments for digital-twin type implementations) and OWC **modulator/demodulator** blocks. We also show results for a combined flowgraph in order to demonstrate how individual blocks can behave as a bottleneck within the complete signal processing chain.

4.1. OWC Channel Model Blocks

A core component of gr-owc is the signal processing blocks for simulating the OWC channel. These blocks account for the optical channel gain along with the electrical to optical and optical to electrical conversion characteristics. The channel gain(s) are dependent on various hardware parameters[2] as well as the relative location and orientation of Tx/Rx pairs. The following channel blocks are implemented[3] in gr-owc.

- **OWC-Channel Model (Relative)**: Models the optical channel by considering the DC channel gain from the Tx to the Rx in relative coordinates.
- **OWC-Channel Model (Absolute)**: Extends the Relative Channel Model by using the absolute coordinates of the Tx and Rx to calculate the relative distance(s) and angles used in the relative block.

The channel model can be parameterized to support multiple transmitters and receivers. The simulated discrete time signal at Rx j is modeled as:

$$y_j[n] = \sum_i x_i[n]H_{ij} + n_j[n] \quad (1)$$

where $x_i[n]$ is the transmitted signal from transmitter i, H_{ij} is the channel gain between transmitter i and receiver j, and $n_j[n]$ is the noise at receiver j. The gain H_{ij} reflects system parameters like hardware characteristics, relative distance(s), and angles. This model accounts for both the transmitted signal and interference at each receiver, along with signal dependent noise, supporting multi-cell and multi-user environments [8].

[2]Detailed descriptions of the functionality of gr-owc blocks can be found in the repository and in prior publications [7–10].

[3]In our evaluation, we used the "Relative" model to assess performance since the real-time functionality of the "Absolute" model is equivalent (i.e., the key difference is the intermittent calculation of angles and distances whenever the absolute positions or orientations are changed).

Comparing the OWC channel model block to other blocks that are core components of the Tx or Rx signal chains, it is important to note that the channel model is not an essential component of a real-time over-the-air implementation (i.e., experimental deployment). When using the channel model block for offline simulation, the throttled rate can be much lower than the modeled sample rate. However, there are certain cases where a channel model is required to keep up with real-time sample rates. As an example, consider an emulation scenario where a simulated channel is used alongside a physical hardware deployment (i.e., scenarios where the simulated channel represents a "digital twin" to co-simulate along with the hardware system under live conditions). This is represented in the flowgraph in Fig. 2 where the OWC channel is highlighted for simulation. The USRP blocks in the flowgraph (shown in Fig. 2 as disabled) can be used to send and receive the waveform in the experimental OWC system, but the channel block must be able to process samples at the same rate as the hardware if both signal paths are intended to operate alongside each other.

4.1.1. Evaluation 1: Coding Implementations

As a modification from the original block implementations in gr-owc, we have enhanced the OWC Channel blocks in gr-owc using VOLK to accelerate per-sample computations by utilizing SIMD instructions. This optimization aims to reduce CPU usage and enable higher sample rate processing suitable for real-time experimentation.

Initially, parallelization was implemented with respect to the number of transmitters, meaning that gain computations from each transmitter were executed in parallel. However, this approach proved inefficient in practice. Because the parallelism was tied to the number of transmitters, fewer transmitters resulted in limited opportunities for parallel execution. Moreover, repeatedly invoking VOLK kernels per transmitter introduced significant overhead, which outweighed the potential benefits, especially with fewer Txs. As a result, the VOLK-based implementation at this stage performed worse than the standard C++ implementation, which was not expected given VOLK's optimization purpose.

To overcome this, the implementation was restructured to parallelize across incoming samples instead. By executing gain and noise computations simultaneously over multiple input samples, the new strategy better utilizes SIMD capabilities. This change enabled greater parallel execution by efficiently utilizing SIMD operations,

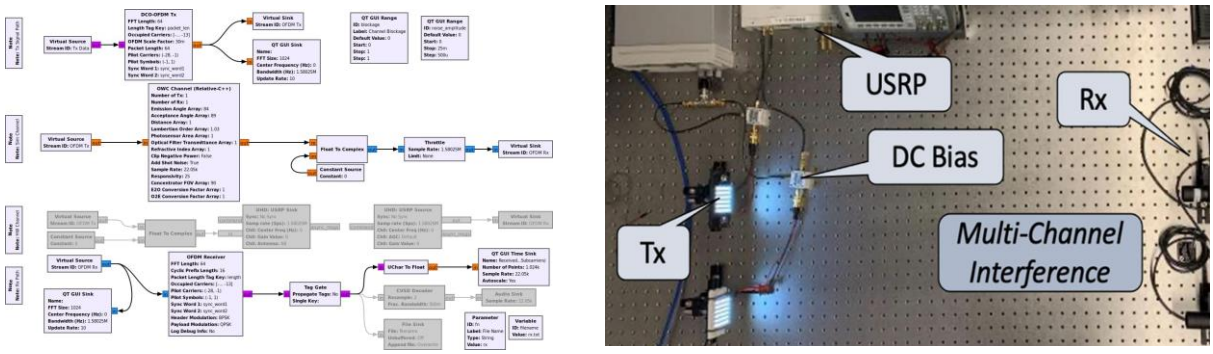

Figure 2: "Digital Twin" model where the OWC channel simulation (left) is run alongside the experimental OWC setup (*right- reproduced from our paper in GNURadio conference 2023).*

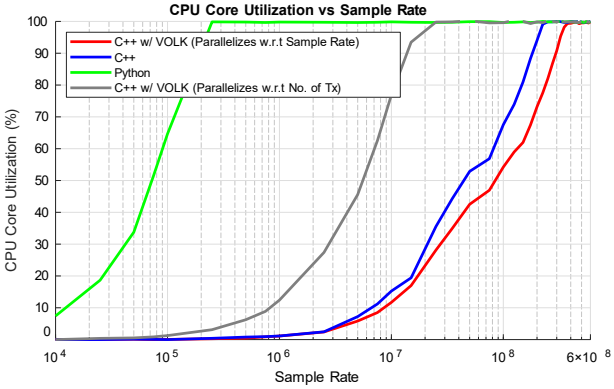

Figure 3: CPU utilization comparison of channel blocks on System 1, implemented in C++ without VOLK, VOLK parallelized at the transmitter level, VOLK optimized at the sample level, and Python, for 1 Tx–1 Rx configurations.

resulting in an improved peak sample rate and more consistent CPU usage under real-time constraints.

As shown in Fig. 3, the Python implementation shows peak CPU utilization at relatively low sample rates. The standard C++ version performs better, but the sample-level VOLK optimization achieves the best efficiency across the full sample rate sweep. In contrast, the transmitter-level VOLK implementation has higher CPU usage than the C++ baseline, resulting in the inefficiency of that parallelization strategy for small numbers of transmitters.

4.1.2. Evaluation 2: Hardware Platforms

Fig. 4 compares different implementations (*C++, C++ with VOLK, and Python*) of the OWC Channel Model across the three systems noted in Table 1. The peak sample rate is defined as the highest input rate a block can process before reaching CPU core utilization saturation, beyond which further increases in sample rate no longer result in higher throughput due to computational limitations.

All three systems have different hardware properties, such as CPU processor (and its clock speed), number of physical cores (and logical thread(s)/core), and other factors that affect the performance of GNURadio blocks. Especially in applications that use a thread-per-block scheduler, such as GNURadio, the number of physical cores rather than logical threads has a significant impact when there are more blocks (threads) running simultaneously. In processors supporting simultaneous multithreading (SMT), an increased number of threads per core can reduce per-thread performance for GNU Radio's thread-per-block execution model, since block threads share the same physical core resources rather than having exclusive access to a core.

In the Intel Core i5, which operates at a maximum clock frequency of 3.20 GHz and has 2 threads per Core, multiple GNU Radio blocks may be scheduled on logical threads that share the same physical core. As a result, each block cannot exclusively utilize the full execution resources of a physical core, leading to reduced per-thread Instructions Per Cycle (IPC) [15,16] due to contention. In contrast, the Raspberry Pi 5 operates at a lower clock frequency of 2.40 GHz but provides four physical cores with a single logical thread per core, allowing each GNU Radio block to execute on a dedicated core and more effectively utilize the available clock resources. Also, the Intel Core i5 represents an older microarchitecture, whereas the Raspberry Pi 5's Cortex-A76 cores benefit from a more modern, energy-efficient microarchitecture optimized for scalar workloads, which aligns well with GNU Radio signal-processing blocks.

Intel Xeon, with 20 CPU cores, shows better performance compared to the other two systems. Designed for high reliability and performance applications, the Intel Xeon achieves a higher sample rate, as seen in Fig. 4. The results of Evaluation 1 are similarly reflected, where C++

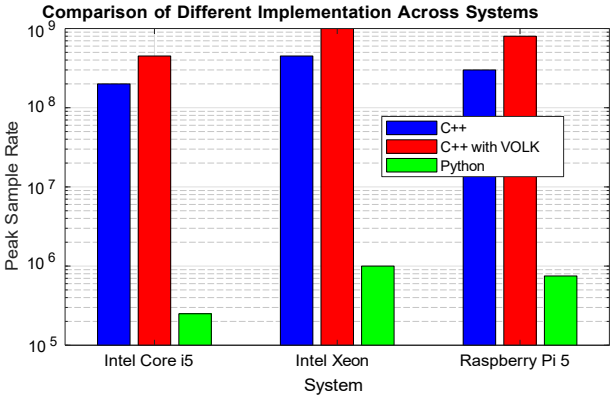

Figure 4: Comparison of different implementations (C++, C++ with VOLK, and Python) of the OWC channel model across different systems, for 1 Tx–1 Rx configurations.

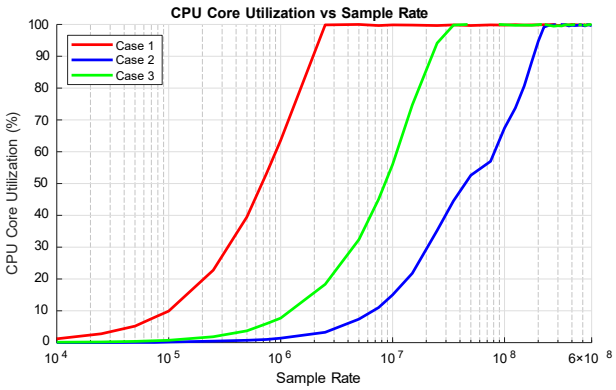

Figure 5: CPU utilization with varying sample rates for Cases 1–3, measured on System 1 with C++ implementation and a 1 Tx–1 Rx configuration.

with VOLK performs better than standard C++, which in turn performs better than Python; and in terms of hardware, Intel Xeon outperforms the Raspberry Pi 5, which in turn performs better than the Intel Core i5. This ordering is consistent with the number of available physical cores, where the Raspberry Pi 5 (4 physical cores) outperforms the Intel Core i5 (2 physical cores). Notably, the performance gains due to going from python to C++ are significantly more impactful than the choice of compute hardware.

4.1.3. Evaluation 3: gr-owc Code Enhancements

In this work, we enhanced the OWC Channel Model implementations in two ways: (1) Core algorithmic optimizations that reduce unnecessary computation, and (2) New features that extend the functionality of channel blocks towards a more realistic representation of a physical OWC link. First, many parameters that influence OWC channel gain, such as Tx and Rx positions, orientations, or physical configurations, tend to remain constant or change infrequently over time. By precomputing the associated channel gain values during block initialization or parameter updates, redundant per-sample calculations are eliminated, reducing the processing load. Second, we introduced features for signal clipping and signal-dependent shot noise to develop a more realistic representation of the OWC channel and enhance simulation accuracy. The impact of these code enhancements are analyzed in the subsequent performance evaluation which observes the following cases.

- **Case 1:** Initially, the channel block calculated the channel gain for every input sample. Since the gain computation involves evaluating parameters like distance, emission angle, and acceptance angle for each Tx-Rx pair, performing this operation repeatedly for every sample led to excessive overhead.
- **Case 2:** The implementation (Case 1) was optimized to precompute the channel gain upon initialization, or whenever a block parameter is modified. Since the gain remains independent of input data or sample, the precomputed values were directly used for processing each input data/sample without recalculating the gain. This modification significantly reduced the overhead caused by repeatedly calculating the gain for each sample.
- **Case 3:** Building on Case 2, a feature was introduced to incorporate shot noise into the model. Shot noise arises due to the quantum nature of photons in optical communication, where the number of detected photons fluctuates randomly, introducing inherent noise in the received signal. This effect is influenced by ambient light and received signal power. While this feature improves the model's accuracy in simulating real-world OWC conditions, it also increases the computational load on the block due to the added noise processing.

Fig. 5 illustrates the CPU utilization trends across different sample rates for all three cases. Case 1 exhibits the highest CPU usage due to per-sample gain computation, whereas Case 2 significantly reduces overhead by precomputing the gain. Case 3 introduces additional computational load due to the signal-dependent shot noise calculations.

4.1.4. Evaluation 4: New Configurations

Recognizing that the newly added features are not relevant in all scenarios (e.g., when operating in the LED's linear

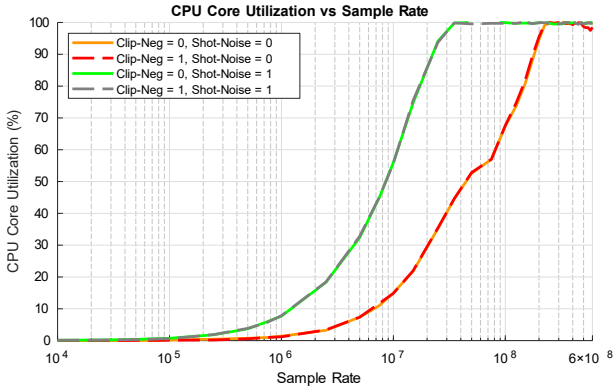

Figure 6: CPU utilization with varying sample rates based on *clip-neg* and *add shot noise* flag states, for 1 Tx–1 Rx configurations.

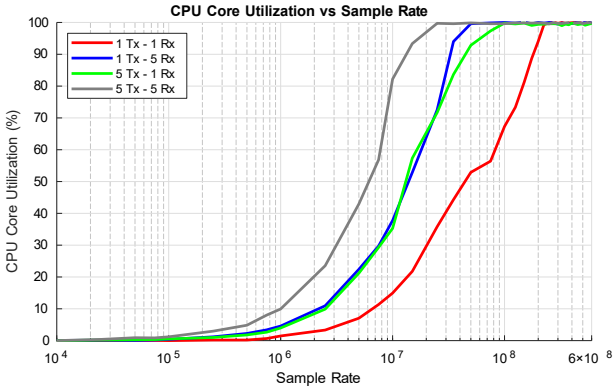

Figure 7: CPU utilization variation for different OWC channel model Tx–Rx configurations.

range or when the noise is dominated by thermal noise making shot noise negligible), we have included configuration flags to make this functionality optional. This allows the features to be disabled in order to reduce computational overhead.

- ***Clip-Neg***: If enabled, this feature removes negative power values from the transmitted signal to ensure that only valid optical power levels are processed.
- ***Add Shot Noise***: If enabled, this feature introduces shot noise at the receiver's end, simulating the signal-dependent shot noise by observing the simulated average optical power received and using this to define the noise power as a function of time.

Each of these features adds additional computation when enabled. Fig. 6 shows how CPU utilization varies with sample rate depending on the state of these flags. When ***clip-neg*** **and** ***add shot noise*** **are disabled** (i.e., Case 2 from Fig. 5), the channel block requires the least computational resources since no extra operations are performed. When **only** ***clip-neg*** **is enabled**, there is not much noticeable increase in CPU utilization. This is because the per-sample check and zeroing of negative values introduce only negligible overhead. When **only** ***add shot noise*** **is enabled**, the computational load increases more significantly compared to *clip-neg*. After computing the received signal from all transmitters, the block must first calculate the average received power for each receiver. This value is then used to generate and add shot noise statistically to each output sample, introducing additional per-sample computation and increasing CPU usage. Lastly, when **both** ***clip-neg*** **and** ***add shot noise*** **are enabled**, CPU utilization is comparable to that of the shot noise only scenario, again due to the negligible impact of clipping.

These results show that while enabling optional features such as clipping and shot noise introduces additional computational load, the overhead associated with checking the corresponding configuration switches is negligible. This justifies the design decision to implement these functionalities within a single configurable block, rather than separating them into multiple specialized blocks. This approach simplifies flowgraph design and improves usability, as researchers can easily toggle optional features without modifying the structure of the flowgraph. Moreover, it offers the flexibility to disable computationally intensive features to achieve higher sample rate requirements when needed.

4.1.5. Evaluation 5: Scaling for Multiple Txs/Rxs

The computational complexity of the OWC Channel Model increases with the number of transmitters and receivers, as more signal processing operations are required. The results from Fig. 7 illustrate how CPU utilization varies based on different Tx-Rx configurations.

- **1 Tx - 1 Rx**: This configuration has the lowest CPU utilization, as only a single transmitter's signal is processed and received by a single receiver.
- **1 Tx - 5 Rx**: The CPU utilization increases because the signal from one transmitter is processed for multiple receivers, requiring additional computational resources.
- **5 Tx - 1 Rx**: This configuration results in similar CPU utilization compared to 1 Tx - 5 Rx since the block must process signals from all five transmitters and combine them at a single receiver. Each transmitted signal undergoes individual channel gain calculations, and all signals must be summed at the re-

ceiver, increasing the overall computational workload.

- **5 Tx - 5 Rx**: This configuration exhibits the highest CPU utilization, as signals from multiple transmitters must be processed and received by multiple receivers, significantly increasing the computational workload.

4.2. OWC Mod/Demod Blocks

Following the same analysis methodology used for evaluating the channel modeling blocks, we consider a set of modulator and demodulator blocks from gr-owc. The following modulation schemes have been implemented in *gr-owc* to support various baseband OWC scenarios:

- **OOK**: Implements the On-Off Keying (OOK) modulation scheme, where binary symbols are represented by high and low signal levels. The OOK Demodulator observes a threshold and defines bits based on the observed samples in a given symbol period.
- **PAM**: Implements Pulse Amplitude Modulation (PAM), assigning varying amplitude levels to symbols, enabling multi-level modulation with multiple bits per symbol. The PAM demodulator averages the received samples over each symbol interval and compares the resulting amplitude against a set of predefined levels to determine the transmitted symbol
- **PPM**: Implements Pulse Position Modulation (PPM), where the position of a pulse within a symbol duration is varied to encode information. The PPM demodulator detects the pulse location within each symbol interval and decodes the symbol based on the identified pulse position.
- **VPPM**: Implements Variable Pulse Position Modulation (VPPM), where binary PPM is combined with adjustable pulse width to enable dimming in dual-use scenarios where the transmitters are used for both communication and illumination. The VPPM demodulator identifies the pulse position to recover the transmitted bit and accounts for pulse width to maintain the desired pulse level.

4.2.1. Evaluation 1: OOK Modulator

By generating multiple samples per symbol (SPS), the OOK Modulator acts as an interpolator block while the OOK Demodulator is a decimator block. For a fixed sample rate, increasing SPS improves signal resolution and can enhance synchronization or filtering accuracy at the cost of reduced bit rate. Alternatively, increasing SPS while maintaining a fixed bit rate leads to a higher sample rate requirement. This also increases the computational load, since more samples must be processed per symbol. These two scenarios can be compared with the different test flowgraphs depicted in Fig. 1. The following tests analyze the effect of SPS and implementation across multiple systems.

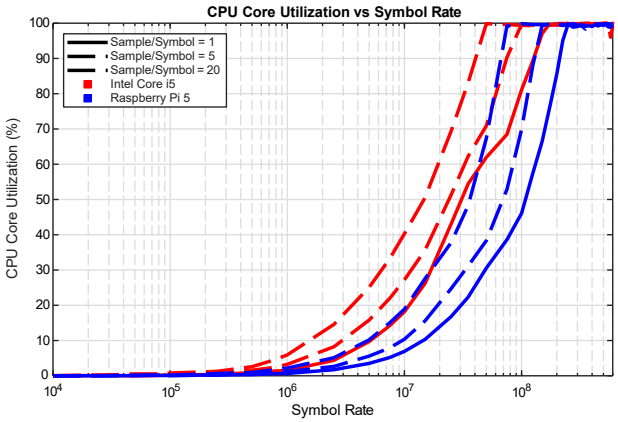

Figure 8: CPU core utilization of OOK modulator (C++) across SPS = 1, 5, 20 on Intel Core i5 and Raspberry Pi 5.

The first test for the baseline OOK block analyzes how SPS settings affect CPU utilization for the OOK Modulator implemented in C++, evaluated on Intel Core i5 and Raspberry Pi 5, with the increase in incoming symbol rate. The SPS values tested were 1, 5, and 20.

As shown in Fig. 8, increasing the SPS consistently raises CPU utilization, as more samples must be processed per symbol period. With the increase in the number of output samples (For SPS >1), the block must process more total samples per second. This leads to higher computational workload, even if the input rate is the same. Since more output is generated, GNU Radio has to schedule the block to run more often to keep up with the downstream blocks' demand, increasing overhead (More samples = more memory writes/reads).

Next, we evaluate the CPU utilization of the OOK Modulator implemented in C++ across three different systems. All systems were tested with SPS = 20 to ensure a consistent load condition. As observed in Fig. 9, the Intel Xeon initially utilizes more computational resources, but beyond a certain symbol rate range, the rate of increase in CPU usage reduces, after which it performs better. The exact reason for the anomalous change in utilization between SPS settings of 20M and 40M is unclear, but it was seen consistently on this computer. The Raspberry Pi 5

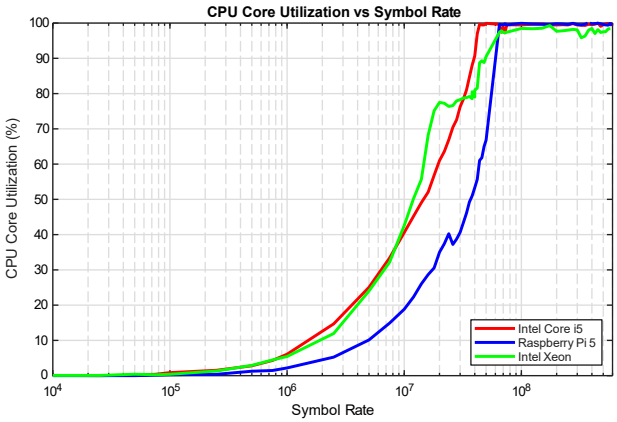

Figure 9: OOK modulator-C++ implementation across Intel Xeon, Intel Core i5, and Raspberry Pi 5 (SPS = 20).

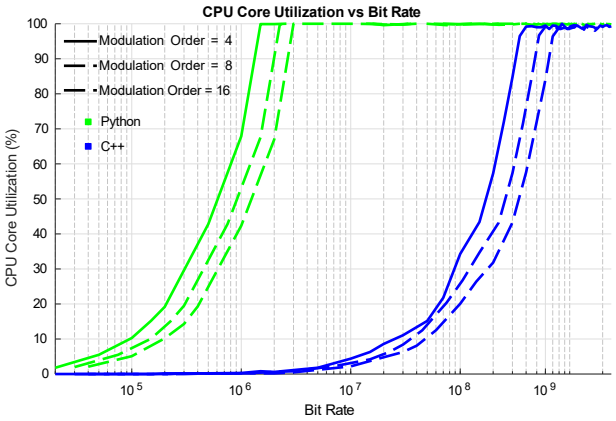

Figure 11: PAM modulator with modulation orders 4, 8, and 16 across C++ and Python implementations.

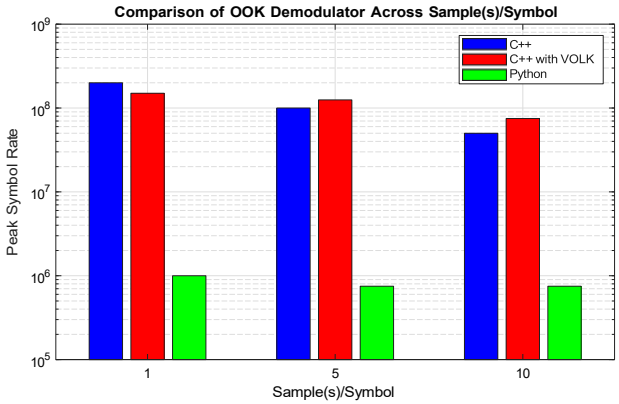

Figure 10: Comparison of OOK demodulator implementations (C++, C++ with VOLK, Python) across different SPS values.

performs better than Intel Core i5, saturating at a higher symbol rate, indicating better efficiency and CPU scaling. Intel Core i5 shows moderate performance but saturates earlier compared to the Pi 5. These results emphasize that while software efficiency is critical, hardware characteristics such as clock speed and number of CPU cores significantly influence real-time processing capabilities.

4.2.2. Evaluation 2: OOK Demodulator

The OOK Demodulator block is the counterpart to the OOK modulator block tested in Evaluation 1. We evaluated the effect of SIMD Optimization on OOK Demodulator performance across SPS and coding implementation (i.e., C++, Python, and C++ with VOLK). The C++ with VOLK implementation was specifically optimized to parallelize operations across samples.

As illustrated in Fig. 10, which was evaluated using flowgraph V2 (Fig.1) at SPS = 1, the C++ implementation outperforms the VOLK-based version. This is due to the absence of parallel execution opportunities at such low SPS, where the overhead of VOLK outweighs any potential gains. However, as SPS increases, the C++ with VOLK implementation begins to outperform standard C++, leveraging parallelism more effectively.

4.2.3. Evaluation 3: PAM Modulator

For PAM implementations, the modulation order (M) relates sample rate and bit rate in a way that is similar to how SPS relates sample rate and symbol rate. Specifically, the modulation order defines the number of amplitude levels used to encode symbols. For example, a modulation order of 4 encodes 2 bits per symbol, 8 encodes 3 bits, and 16 encodes 4 bits per symbol. The bit rate is the rate at which bits are transmitted and is determined by both the sample rate and the modulation order. To evaluate the computational efficiency of the PAM Modulator, we tested both the Python and C++ implementations across varying bit rates and modulation orders (4, 8, and 16) with SPS = 1, on system 3. The CPU utilization patterns for both implementations are illustrated in Fig. 11.

As the modulation order increases, the number of bits per symbol also increases. This allows the block to transmit the same bit rate using fewer symbols per second. This also implies that a fixed SPS setting across the modulation orders leads to a lower sample rate for higher modulation orders. As a result, the block processes fewer samples per unit time, leading to lower CPU utilization with higher modulation orders.

4.2.4. Evaluation 4: Comparison across Modulators

In this evaluation, four gr-owc modulators (OOK, PAM, PPM, and VPPM) are evaluated using flowgraph V2 (Fig.1). All modulators were run at SPS = 32 with M = 16 for PAM and PPM. At SPS = 32, OOK, VPPM, PAM, and PPM operate at the same sample rate and symbol rate, but differ in bit rate due to differences in modulation order. We additionally tested the SPS = 8 configurations of OOK and VPPM which operate at the same sample rate and bit rate as the SPS = 32 PAM and PPM configurations with M = 16, while using different symbol rates. These two operating points allow us to separate the effects of sample rate, symbol rate, and modulation complexity on CPU utilization.

From Fig. 12, the observed trends can be directly attributed to implementation-level design choices. In the OOK modulator, each output sample is generated dynamically based on the incoming symbol stream, resulting in per-sample decision logic and repeated value assignment. In contrast, the VPPM implementation pre-computes complete samples-per-symbol waveforms and copies them to the output buffer using memory copy operations. This approach significantly reduces per-sample computation and explains the lower CPU utilization observed for VPPM relative to OOK at comparable sample rates. As future work, the OOK modulator could be refactored to use a similar pre-initialized waveform (array) and memory copy strategy, which is expected to further reduce its CPU overhead and improve performance.

An important observation from Fig. 12 is that when CPU utilization is viewed only as a function of sample rate, lower-order modulation schemes such as OOK and VPPM appear more efficient, showing lower CPU usage at comparable sample rates. However, this view changes when bit rate is considered as the primary metric. The SPS = 8 OOK and VPPM configurations operate at the same sample rate and bit rate as the SPS = 32 PAM and PPM configurations with M = 16, but require a higher symbol rate. This higher symbol rate increases the frequency of symbol-level processing, reducing the apparent efficiency of OOK and VPPM when normalized by delivered bit rate. When compared at equivalent bit rates, the higher-order modulation schemes such as PAM and PPM provide better efficiency of CPU utilization. This trend is also consistent with the results shown in Fig. 11, where higher-order modulation schemes perform better when evaluated based on achievable bit rate rather than sample rate alone.

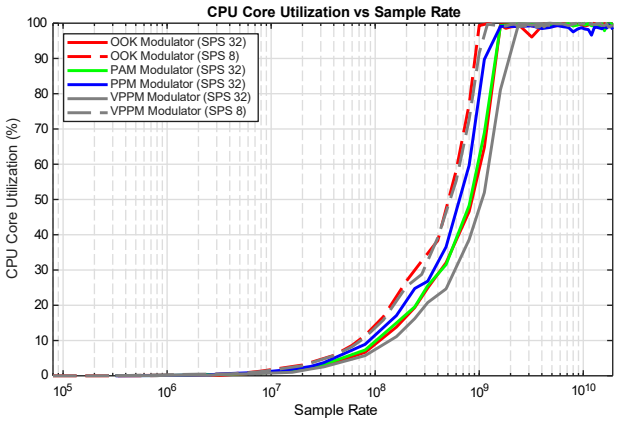

Figure 12: CPU utilization comparison across all modulation schemes (OOK, PAM, PPM, VPPM) on Raspberry Pi 5.

For the PAM modulator, multiple samples-per-symbol arrays are pre-initialized corresponding to each of the M = 16 amplitude levels. During runtime, each input symbol must be compared against all possible amplitude levels to determine the correct waveform before copying the corresponding array to the output buffer. This additional symbol-level comparison and indexing overhead results in higher CPU utilization than OOK and VPPM, as reflected in the measured results.

The PPM modulator incurs the highest computational cost due to its waveform construction strategy. For each symbol, the entire samples-per-symbol region is first initialized to the minimum magnitude, after which the pulse position is calculated and selectively overwritten with maximum magnitude values. This two-stage initialization process introduces additional per-symbol operations and memory writes, leading to the highest CPU utilization among all evaluated modulators.

Overall, these results highlight how waveform initialization strategies and memory access patterns play a critical role in determining the computational efficiency of signal processing blocks. Even at identical sample rates, differences in symbol processing logic and data movement can lead to substantial variations in CPU utilization.

4.2.5. Evaluation 5: Comparison across Demodulators

In this evaluation, four gr-owc demodulators (OOK, PAM, PPM, and VPPM) are evaluated using the V1 test (Fig. 1). All demodulators were tested at SPS = 32, with a modulation order of M = 16 for PAM and PPM. Similar to the modulator evaluation, OOK and VPPM demodulators were additionally evaluated at SPS = 8 to enable fair comparisons

under both sample-rate-matched and bit-rate-matched conditions.

From Fig. 13, the OOK demodulator shows the lowest CPU core utilization across all evaluated demodulators. OOK demodulation consists of averaging the incoming samples over each symbol interval, followed by a single threshold comparison to decide between the two possible transmitted symbols (ON or OFF). This simple processing pipeline results in minimal per-symbol computation.

The VPPM demodulator exhibits higher CPU utilization than OOK but remains significantly more efficient than PAM and PPM. VPPM demodulation uses a maximum-likelihood decision process between two possible symbol waveforms. Pre-initialized reference waveforms are correlated with the received samples, and the symbol corresponding to the higher correlation metric is selected. While this matched-filter operation introduces additional per-symbol computation compared to OOK, the decision space is limited to two symbols, keeping the overall computational cost relatively low.

At a fixed sample rate, the 8-SPS OOK and VPPM demodulators exhibit higher CPU utilization than their respective 32-SPS configurations. Although both configurations process the same number of samples per second, the lower SPS results in a higher symbol rate, causing symbol-level demodulation operations to be executed more frequently. This increased symbol processing rate leads to higher overall CPU utilization for the 8-SPS implementations.

PPM demodulation requires evaluating the received samples against all possible pulse-position waveforms within each symbol interval. For M = 16, this involves computing correlation metrics for 16 candidate pulse positions and selecting the most likely symbol using a maximum-likelihood process. The combination of full-symbol waveform initialization, repeated correlation operations, and multi-symbol comparison results in substantially higher computational overhead compared to VPPM, and OOK.

Unlike OOK demodulation, which relies on a single threshold decision, PAM demodulation also begins by averaging samples over each symbol interval; the averaged value must be compared against multiple amplitude levels corresponding to the modulation order. For M = 16, this requires evaluating the received symbol against 16 possible levels to determine the transmitted symbol, introducing additional comparison and decision overhead. This multi-level decision and averaging process accounts for the increased CPU utilization observed for PAM relative to all the demodulators.

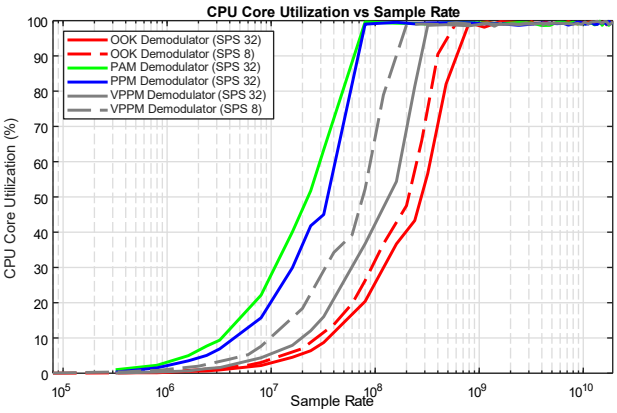

Figure 13: CPU utilization comparison across all demodulation schemes (OOK, PAM, PPM, VPPM) on Raspberry Pi 5.

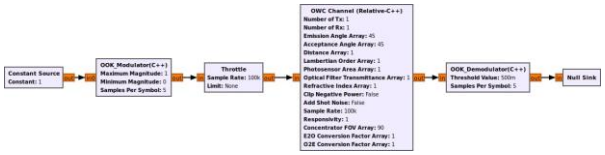

Figure 14: OWC flowgraph used in complete system evaluation.

4.3. OWC System (Complete Flowgraph)

The complete OWC system flowgraph shown in Fig. 14 represents a baseline configuration for end-to-end performance evaluation of gr-owc. The results in Fig. 15 compare the full system *1 Tx - 1 Rx* configuration, tested on Intel Core i5 with all blocks implemented in C++, with the baseline unit test results for the individual blocks.

The results show that the OOK Demodulator reaches saturation at a lower sample rate compared to other blocks. As sample rate increases, the demodulator becomes the first block to reach maximum CPU capacity. Consequently, the entire complete flowgraph (black curve) also flattens at that point, indicating a bottleneck imposed by the OOK Demodulator. While the OOK Modulator and Channel Model exhibit continued scalability with increasing sample rates, they are ultimately limited by the demodulator's processing capacity in the full system flowgraph.

The overhead in the graph (gray curve) is the combined CPU usage of the Constant Source, Throttle, and Null Sink blocks, which perform minimal computation. In the image, the complete flowgraph reaches ≈280% CPU utilization, meaning that the flowgraph is effectively consuming the equivalent of ≈2.8 CPU cores, indicating its multi-core usage behavior. This highlights a critical insight in system-level OWC SDR design: even if individual

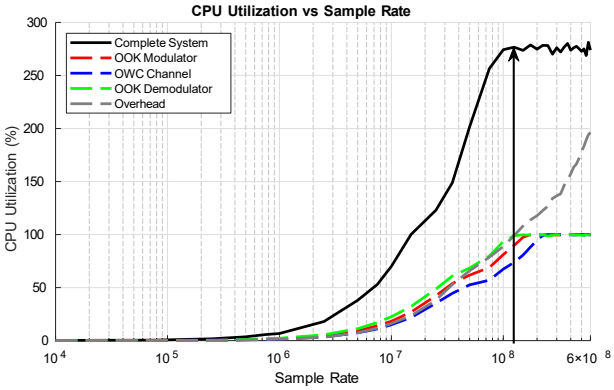

Figure 15: CPU utilization of individual blocks and complete system flowgraph tested on Intel Core i5 using C++ blocks.

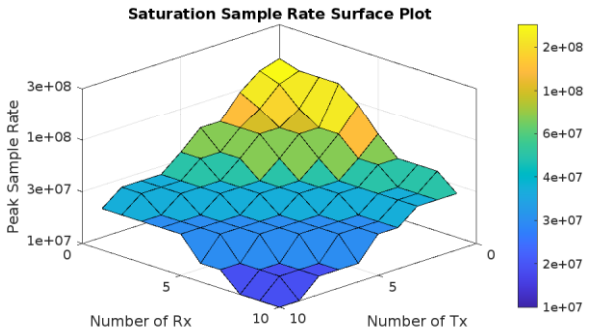

Figure 16: CPU utilization of the complete flowgraph under different transmitter (Tx) and receiver (Rx) configurations, implemented using C++ blocks.

upstream blocks are capable of handling higher throughput, the block with the highest computational load (here, the demodulator) limits the overall system performance. Identifying and optimizing such bottlenecks is essential for high-throughput real-time processing in SDR systems.

The surface plot in Fig. 16 shows the peak sample rate of a complete OWC system flowgraph as a function of the number of transmitters and receivers. In this configuration, each transmitter is paired with a dedicated Constant Source, Throttle, and OOK Modulator block, while each receiver is paired with an OOK Demodulator and Null Sink. All transmitters and receivers are connected to a shared OWC Channel Model, which is responsible for handling the signal transmission between all Tx-Rx pairs.

Each point on the surface plot corresponds to the saturation sample rate before the system reaches max CPU utilization capacity. As the number of transmitters and receivers increases, the computational demand on the Channel Model block also increases due to the higher number of gain calculations and signal summations required for each additional link. This results in a general decline in the peak sample rate for larger Tx-Rx configurations. The plot highlights the scalability limitations of the system and shows that increasing the number of nodes imposes greater computational usage.

5. Conclusions

This work presents a comprehensive performance characterization of the gr-owc, GNURadio OOT module for optical wireless communication systems, emphasizing real-time processing capabilities across diverse hardware configurations and performance impacts based on implementation methods, which could be valuable for other researchers to understand when developing custom GNU-Radio blocks/modules. Through systematic evaluation of individual signal processing blocks and complete system flowgraphs, we analyze CPU utilization trends concerning key operational parameters such as sample rate, modulation order, and samples per symbol.

The results show that C++ implementations consistently outperform Python in computational efficiency, with additional gains achieved through SIMD optimization using VOLK, particularly in high-sample workloads. C++'s nature as a compiled language allows for faster execution speeds compared to Python, which is interpreted. This speed advantage is crucial in applications like SDR where real-time operation requires continuous processing of an incoming sample stream.

Channel model analysis revealed that configuration factors like the number of transmitters and receivers significantly influence CPU usage, and precomputing static channel parameters is beneficial for reducing overhead. Cross-platform benchmarking further highlights the role of hardware characteristics, particularly clock speed, memory access, and core count, sustaining high-throughput real-time operation.

In complete OWC system flowgraph evaluations, it was observed that once any individual block reaches its maximum computational capacity, it imposes a bottleneck on the entire system. This prevents downstream or upstream blocks from utilizing higher sample rates, effectively limiting overall throughput. Identifying and optimizing such limiting blocks is essential for achieving scalable real-time performance in SDR systems.

Overall, the study shows that gr-owc is a practical and flexible research platform for real-time optical wireless experimentation, making it well-suited for building SDR-based OWC systems with multi-user / multi-cell configurations. Given the complexity of dense multi-no

environments, experimental proof-of-concept testing is a key component to understanding the performance of novel protocols and signal processing chains. The tools within gr-owc can be combined with the broader functionality of GNURadio to support a simulation-to-experimentation workflow for OWC network testing. Accordingly, the low- level blocks can be further developed into system models for VLC networks, VLP systems, and hybrid radio/optical networks - allowing for practical evaluation in different environments and offering real-world validation of sys- tem performance metrics. The performance analysis pro- vided in this work demonstrates the limits of individual blocks, while also providing a structure for performance analysis of more complex flowgraphs. As complexity is added to develop system functionality, awareness of bot- tlenecks within the signal chain will highlight where at- tention is needed in order to optimize end-to-end system performance.

List of Abbreviations

- IPC: Instructions Per Second
- LED: Light Emitting Diode
- OOK: On-Off Keying
- OOT: Out-of-Tree
- OWC: Optical Wireless Communication
- PAM: Pulse Amplitude Modulation
- PPM: Pulse Position Modulation
- Rx: Receiver
- SDR: Software Defined Radio
- SIMD: Single Instruction, Multiple Data
- SMT: Simultaneous Multithreading
- SPS: Samples Per Symbol
- Tx: Transmitter
- UUT: Unit Under Test
- VLC: Visible Light Communication
- VOLK: Vector Optimization Library of Kernels
- VPPM: Variable Pulse Position Modulation

Funding
This work was partially supported by the National Science Foundation (NSF) ERI Program under Grant No. 2347514, and by the University of Massachusetts Boston's Healey Research Grant Program.

Authors' Contributions:
Conceptualization, Funding Acquisition, Project Administration, Resources, Supervision: M.R.; Data Curation, Software, Investigation, Visualization, Writing – Original Draft: K.S.; Methodology, Validation, Formal Analysis, Writing – Review & Editing: M.R. and K.S. All authors have read and agreed to the published version of the manuscript.

ORCID IDs:
Kunal P. Sangurmath: 0009-0009-4414-6128
Michael B. Rahaim: 0000-0002-7205-1929

Data and Materials Availability Statement
All data and code supporting this study are available at growc GitHub repository (examples/perf-data). Reproduction instructions are provided in the README.md file within the full repository [17].

Conflict of Interest
None of the authors has any conflicts of interest to disclose concerning this study.

AI Declaration:
ChatGPT was used in a limited capacity to support editorial refinement of select text passages. The intellectual framework, proposal structure, technical methodology, and research contributions were conceived and developed independently by the authors. All content has been reviewed in full to ensure accuracy, technical integrity, and compliance with proposal requirements.

Acknowledgement:
The authors gratefully acknowledge partial support for this work from the National Science Foundation (NSF) ERI Program under Grant No. 2347514, and from the University of Massachusetts Boston's Healey Research Grant Program.

References

[1] Abdalla I, Rahaim M, Little T. Interference in multiuser optical wireless communications systems. Philosophical transactions of the Royal Society of London Series A: Mathematical and physical sciences. 2020 03;378.

[2] Al-Ahmadi S, Maraqa O, Uysal M, Sait SM. MultiUser Visible Light Communications: State-of-the-Art and Future Directions. IEEE Access. 2018;6:70555- 71.

[3] Demir MS, Sait SM, Uysal M. Unified Resource Allocation and Mobility Management Technique Using Particle Swarm Optimization for VLC Networks. IEEE Photonics Journal. 2018;10(6):1-9.

[4] Obeed M, Salhab AM, Alouini MS, Zummo SA. On Optimizing VLC Networks for Downlink Multi-User Transmission: A Survey. IEEE Communications Surveys & Tutorials. 2019;21(3):2947-76.

[5] Yang L, Zhang Q, Zhang W, Deng L, Yang H. Resource Allocation for Hybrid Visible Light Communications (VLC)-WiFi Networks. IEEE Access. 2020;8:176588-97.

[6] Blossom E. GNU radio: tools for exploring the radio frequency spectrum. Linux J. 2004 Jun;2004(122):4.

[7] Ahmed A, Rahaim MB. gr-owc: An Open Source GNURadio-Based Toolkit for Optical Wireless Communications. In: 2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob); 2021. p. 48-53.

[8] Ahmed A, Dzhezyan G, Aboutahoun H, Chu V, DiViccaro J, Ohanian V, et al. SDR Beyond Radio: An OOT GNU Radio Library for Simulation and Deployment of Multi-Cell / Multi-User Optical Wireless Communication Systems. Proceedings of the GNU Radio Conference. 2022;7(1). Available from: https://pubs. gnuradio.org/index.php/grcon/article/view/118.

[9] Onwuchekwa C, AminuMukhtar M, Martinez L, Lemus A, Planchart V, Semash F, et al. Design and Implementation of a Multi-Node Optical Wireless Communication Testbed for Centralized Configuration and Adaptation of System Parameters using GNU Radio's XML-RPC and ZMQ Modules. Proceedings of the GNU Radio Conference. 2023;8(1). Available from: https://pubs.gnuradio.org/index.php/grcon/ article/view/139 .

[10] Rahaim M, Ahmed A, Kulhandjian M, Kulhandjian H. Optical OFDMA for Multi-Cell/Multi-User

Indoor Optical Wireless Networks. In: 2024 58th Asilomar Conference on Signals, Systems, and Computers; 2024. p. 742-7.

[11] Orhan D, Lima Pilla L, Barthou D, Cassagne A, Aumage O, Tajan R, et al. Optimal scheduling algorithms for software-defined radio pipelined and replicated task chains on multicore architectures. Journal of Parallel and Distributed Computing. 2025;204:105106. Available from: https://www.sciencedirect.com/ science/article/pii/S0743731525000735 .

[12] AlShekh RH, Dawwd SA, Qassabbashi FN. Comparative Review of Multicore Architectures: Intel, AMD, and ARM in the Modern Computing Era. Chips. 2025;4(4). Available from: https://www.mdpi.com/2674-0729/4/4/44 .

[13] Stoico V, Dragomir AC, Lago P. An Empirical Study on the Performance and Energy Usage of Compiled Python Code. In: Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering. EASE '25. New York, NY, USA: Association for Computing Machinery; 2025. p. 46–56. Available from: https://doi.org/10.1145/3756 681.3756972 .

[14] Laskos C, Zubow A, Dressler F. Latency Analysis of SDR-based Experimental C-RAN / O-RAN Systems. In: 2025 IEEE International Conference on Communications Workshops (ICC Workshops); 2025. p. 893-8.

[15] Abad P, Prieto P, Puente V, Gregorio JA. Efficient Server Consolidation through a balanced mix of Transformer-based and Conventional Applications. In: Proceedings of the 39th ACM International Conference on Supercomputing. ICS '25. New York, NY, USA: Association for Computing Machinery; 2025. p. 764–775. Available from: https://doi.org/10.1145/3721145.3725751 .

[16] Sari S, Nezir U, Demir O, Kucuk G. Breaking the Complexity Barrier: Enhancing Quality of Service in Simultaneous Multithreading Processors. Array. 2025;28:100602. Available from: https://www.sciencedirect.com/science/article/pii/S2 590005625002292 .

[17] Ahmed A, Sangurmath K, Rahaim M. gr-owc: An Open Source GNURadio-Based Toolkit for Optical Wireless Communications. 10.5281/zenodo.5237300; 2026. https://github.com/UCaNLabUMB/gr-owc.