



Performance Characterization of *gr-owc*, the GNU Radio 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 02125, USA

² Engineering Department, University of Massachusetts, Boston, MA 02125, USA

Article History

Received: August 29, 2025

Accepted: May 11, 2026

Published: May 26, 2026

Abstract

As researchers continue to investigate novel technologies for next-generations of wireless communications systems, flexible and adaptable tools remain essential for system testing and development. Software-Defined Radios (SDRs) provide flexible and reconfigurable signal-processing capabilities, thereby serving as a crucial platform for the rapid prototyping, testing, and experimental validation of communication systems. GNU Radio is an open-source SDR toolkit that enables the extension of its core software library through the development of Out-of-Tree (OOT) modules. The proposed *gr-owc* OOT module provides SDR capabilities for Optical Wireless Communication (OWC) and serves as a research platform for real-time experimentation, system-level analysis, and algorithm development for OWC systems. To support real-time experimentation effectively, the custom signal-processing blocks implemented in *gr-owc* must be computationally efficient to meet real-time hardware constraints. This paper presents a comprehensive analysis of the Central Processing Unit (CPU) utilization patterns associated with *gr-owc* signal-processing blocks. Specifically, the study provides insights into the computational efficiency of *gr-owc* blocks and representative flowgraphs by analyzing CPU utilization across sample rate, hardware deployment platforms, and block configurations. In addition, recent modifications to these blocks are introduced to enhance performance while preserving their original functionality. Experimental results demonstrate the influence of factors such as computational hardware and code implementation on CPU utilization and further reveal the limitations of single-core block processing in GNU Radio applications.

Keywords:

Optical Wireless Communication (OWC); Software Defined Radio (SDR); Visible Light Communication (VLC); GNU Radio; CPU utilization; real-time signal processing

1. Introduction

OWC has attracted significant interest from the research community in recent years, initially focusing on point-to-point links and the development of 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 operating in multi-cell environments or in cells shared by multiple users face additional challenges, including dynamic resource allocation, handover manage-

ment, and interference mitigation [1,2]. Novel algorithms have been proposed and evaluated theoretically and through simulation studies [3–5]; however, experimental validation of techniques for multi-node OWC systems remains limited. This limitation is primarily attributed to the complexity of developing proof-of-concept implementations that accurately capture interference, mobility, and real-world deployment constraints, while simultaneously enabling precise control at the signal level. To address this challenge, software-defined radio (SDR) tools and

* Corresponding Author:

Michael B. Rahaim, Engineering Department, University of Massachusetts, Boston, MA 02125, USA, michael.rahaim@umb.edu; Tel.: +1-617-287-4132



© 2026 Copyright by the Authors.

Licensed as an open access article using a [CC BY 4.0 license](https://creativecommons.org/licenses/by/4.0/).

techniques are leveraged for OWC research. In particular, the GNU Radio SDR toolkit is employed, incorporating custom software and hardware modifications to support OWC signal processing.

GNU Radio is a widely used SDR toolkit, valued for its accessibility and flexibility in RF research and experimental validation [6]. While GNU Radio provides a comprehensive library of signal-processing tools, it also enables the development of custom Out-of-Tree (OOT) modules that extend its core functionality. Previously, the *gr-owc* OOT module was introduced to support OWC research [7–10]. It provides a comprehensive suite of custom signal processing blocks, detailed documentation, and representative sample flowgraphs, enabling real-time signal processing experimentation, system-level performance analysis, and algorithm development for multi-node OWC networks. The signal processing blocks and sample flowgraphs serve as a “golden reference” for evaluating novel techniques and improvements, thereby supporting further innovation in the field.

Key advantages of SDR include its ability to support simulation-to-experimentation workflows and its high degree of configuration flexibility. In particular, SDR enables researchers to validate proposed techniques in simulated environments prior to deployment in experimental setups, requiring only minimal modifications for real-world implementation. In addition, its flexibility enables the interchange of front-end transmitter (Tx) and receiver (Rx) configurations while preserving common signal processing chains and test flowgraphs, thereby ensuring consistency across different experimental setups. The *gr-owc* module extends the advantages of SDR to OWC research, bridging the gap between simulation and real-world experimentation. It provides a cost-effective and flexible alternative to traditional OWC hardware, which is often expensive and inflexible. By leveraging *gr-owc* and the broader GNU Radio framework, researchers can validate novel OWC techniques in simulated environments and seamlessly transition to experimental implementations without requiring significant modifications.

Compared to offline simulation tools, real-time SDR-based test systems enable experimental setups that closely emulate practical operating scenarios and facilitate the validation of system designs under realistic operational constraints. However, they also require significant attention to potential processing bottlenecks. This is particularly evident when signal processing is implemented on a general-purpose processor. SDR platforms such as GNU Radio provide a robust and flexible framework for implementing and evaluating signal processing methods in both simulated and real-world experimental environments. However, the performance limits of individual

blocks must be understood prior to deployment in experimental systems to identify potential bottlenecks. Characterizing these bottlenecks enables determination of where targeted optimizations are required, potentially including FPGA offloading (e.g., with GNU Radio’s RFNoC tools).

In this work, the processing blocks of *gr-owc* are evaluated for their ability to meet real-time processing requirements. The evaluation focuses on quantifying the performance consistency of *gr-owc* blocks across different hardware configurations and software implementations. Performance improvements resulting from recent code modifications are also demonstrated across various blocks within the *gr-owc* module. While prior work in [7–10] established baseline functionality, the present study highlights targeted optimizations that enhance 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 GNU Radio.

The remainder of this paper is organized as follows. **Section 2** provides background on SDR, GNU Radio, and the *gr-owc* module. **Section 3** describes the experimental test procedures and methodologies used to measure CPU utilization. **Section 4** presents the test configurations and evaluates the results. **Section 5** concludes the paper by summarizing the key findings.

2. Background

SDR has emerged as a key enabler of configurable RF communication systems. Current SDR research often emphasizes the capability to implement dynamic and adaptive system functionalities. However, the ability to rapidly translate ideas from theory or simulation into practical implementations represents another important aspect of the “software-defined” paradigm. This design flexibility has proven highly valuable to the research community. By lowering the barrier to entry and enabling practical implementation of novel concepts, SDR has expanded access to experimental research in wireless communications and promoted a more inclusive research environment.

2.1. The GNU Radio SDR Toolkit

SDRs provide a flexible platform for experimental validation of communication systems, and GNU Radio is one of the most widely used toolkits for developing SDRs. In GNU Radio, signal processing blocks can be interconnected within a flowgraph to construct custom signal processing chains. GNU Radio also provides support for the development of custom blocks and out-of-tree (OOT) modules, enabling extension of its core function-

ality. These OOT modules exist outside the GNU Radio source tree. OOT modules are developed to extend GNU Radio with custom functionality and signal processing blocks while maintaining an independent codebase separate from the core framework. These modules can be selectively installed by users as required. This approach facilitates the development of additional capabilities alongside the main GNU Radio distribution and enables efficient sharing of functionality with the broader research community.

2.2. The *gr-owc* OOT Module

The *gr-owc* OOT module [7] extends the capabilities of GNU Radio to support OWC techniques by providing custom signal processing blocks and flowgraphs tailored for OWC research applications. 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 a range of modulation and demodulation techniques, as well as a configurable OWC channel model. The seamless integration with GNU Radio offers flexibility for diverse flowgraphs that combine *gr-owc* blocks with signal processing blocks from the core GNU Radio library. Additionally, it supports real-time signal processing and incorporates hardware-related considerations for applying SDR concepts to OWC, thereby facilitating the transition from simulation-based studies to experimental validation [8–10].

3. Materials and Methods

To evaluate the real-time performance of blocks within *gr-owc*, a common flowgraph structure was implemented to isolate block-level performance and analyze CPU utilization as a function of sample rate. A representative set of *gr-owc* blocks was evaluated across multiple hardware platforms and software configurations to assess their impact on real-time processing performance and to identify potential computational bottlenecks.

3.1. Test Procedure

To evaluate the *gr-owc* blocks, the testing setup follows standardized flowgraphs, as shown in Figure 1, to isolate the performance of a given block as the 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, two versions of the test flowgraph are considered. With the OOK modulator, flowgraph V1 evaluates performance with respect to bit rate, whereas V2 evaluates performance with respect to sample rate. 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 introducing significant computational 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.

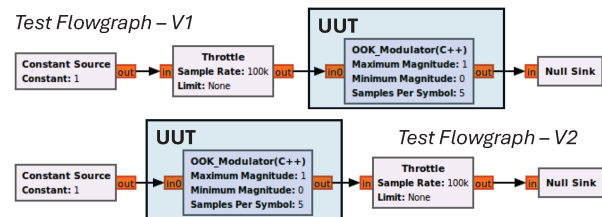


Figure 1: Example performance testing flowgraphs with *gr-owc*'s On-Off-Keying modulator block as the UUT.

To analyze CPU utilization trends, the test systematically sweeps across different sample-rate values¹, measuring CPU usage at each step. For each sample rate, CPU utilization is recorded using the *pidstat* tool by monitoring the process ID of the GNU Radio process. A total of 15 CPU utilization readings were collected for each sample rate, and the results were averaged to obtain a stable measurement. GNU Radio employs a thread-per-block scheduler [11], in which each block executes in its own thread and cannot use more than one CPU core, thereby restricting each block to single-core execution. However, processing across different blocks within a flowgraph can be distributed across multiple CPU cores.

The Throttle block regulates the overall flow of samples by enforcing an upper limit on the sample rate; however, it does not impose a minimum sample rate. If downstream blocks are unable to keep up with the specified rate, the sample flow is correspondingly reduced throughout the flowgraph. Therefore, when the CPU utilization

¹Here, 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.

saturates (i.e., stops increasing with respect to the throttle's sample rate parameter), it indicates that the processing capacity of the UUT, rather than the configured sample rate of the throttle block, has become the primary bottleneck. This is defined as the peak sample rate for a given block under a specific test scenario.

Knowledge of the peak sample rate is critical for determining whether the unit under test (UUT) can process incoming samples at the specified rate or whether CPU constraints prevent real-time operation, resulting in processing delays or dropped samples. This analysis enables assessment of the real-time processing capabilities of *gr-owc* blocks and their scalability as sample rates increase. It also provides insight into the ability of a given flowgraph to sustain target hardware sample rates

when transitioning to experimental test cases, where the sample rate is governed by SDR hardware rather than a Throttle block.

3.2. Compute Hardware

The *gr-owc* processing blocks were tested on three different compute systems (Table 1) to evaluate the impact of compute hardware characteristics on performance [12]. Since a major focus of *gr-owc* is to enable low-cost proof-of-concept experimentation for OWC systems, performance is evaluated across a range of hardware platforms, from a Raspberry Pi single-board computer to a higher-performance multi-core system. For all test systems, Ubuntu 22.04.3 LTS was installed.

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

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; as a result, both threads share the same core resources, which can reduce the effective instructions per cycle (IPC) for each thread when executed concurrently. System 2 (Intel Xeon Gold 6230) provides multiple physical cores, each with a single thread, allowing each thread to execute independently and achieve higher, more consistent IPC. System 3 (Raspberry Pi 5) also employs a one-thread-per-core configuration; however, it features a lower number of cores and a reduced clock frequency. In general, systems with a higher number of physical cores and a single thread per core provide more consistent IPC per thread, whereas systems with multiple threads per core trade per-thread IPC for higher overall resource 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 trade-offs in terms of development complexity and real-time performance, allowing developers to select the implementation strategy that best aligns with their performance and development requirements.

- **Python:** Ideal for non-time-critical tasks due to its ease of use and faster development cycles. 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* have primarily focused on functional implementation, with blocks predominantly developed in Python. In this work, real-time performance improvements are demonstrated through the recent development of C++ implementations for components within the *gr-owc* module. This work is also based on the *gr-owc* codebase, which has been updated for compatibility with GNU Radio Release 3.10, whereas prior publications considered an earlier version of *gr-owc* 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] when higher sample rates are transmitted to the hardware. Second, the communication interface between the host system and the USRP, such as Ethernet, imposes bandwidth limitations on achievable bit rates 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 assesses whether the system processor can sustain real-time operation under progressively increasing sample rates. This approach enables the isolation of the computational performance of the *gr-owc* blocks in terms of real-time feasibility.

To provide a detailed analysis of specific blocks within *gr-owc*, a focused subset of components is considered. Specifically, the OWC channel modeling blocks—considered in this analysis for the purpose of real-time simulation in conjunction with experimental deployments when applied to digital-twin-based implementations—and the OWC modulator/demodulator blocks are analyzed. Results are also presented for a combined flowgraph to demonstrate how individual blocks may act as bottlenecks within the complete signal processing chain.

4.1. OWC Channel Model Blocks

A core component of *gr-owc* consists of signal processing blocks designed for OWC channel simulation. These blocks account for the optical channel gain, and the electro-to-optical and opto-to-electrical conversion characteristics. The channel gains depend on various hardware parameters, and on the relative position and orientation of the transmitter–receiver (Tx/Rx) pairs. The following channel blocks are implemented 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 follows:

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

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, as well as signal-dependent noise, thereby supporting multi-cell and multi-user environments [8].

Comparing the OWC channel model block with other core components of the Tx or Rx signal chains, it is important to note that the channel model is not an essential component in real-time over-the-air implementations, i.e., experimental deployments. 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 in which a simulated channel is used alongside a physical hardware deployment, where the simulated channel serves as a digital twin to enable co-simulation with the hardware system under live operating conditions. This is illustrated in the flowgraph in Figure 2, where the OWC channel is highlighted to indicate its role in simulation. The USRP blocks in the flowgraph (shown in Figure 2 as disabled) can be used to transmit and receive waveforms in the experimental OWC system; however, the channel block must be capable of processing samples at the same rate as the hardware when both signal paths are intended to operate concurrently.

4.1.1. Evaluation I: Coding Implementations

As a modification to the original block implementations in *gr-owc*, the OWC channel blocks have been enhanced with VOLK to accelerate per-sample computations by leveraging SIMD instructions. This optimization reduces CPU utilization and enables higher sample-rate processing, thereby supporting real-time experimentation.

Initially, parallelization was implemented across the number of transmitters, such that channel gain computations for each transmitter were executed in parallel. However, this approach proved inefficient in practice. Since the degree of parallelism depended on the number of transmitters, configurations with fewer transmitters provided limited opportunities for concurrent execution. Moreover, repeated invocation of VOLK kernels per transmitter in-

roduced significant overhead that outweighed the potential benefits, particularly in scenarios with a small number of transmitters. As a result, the VOLK-based implementation at this stage exhibited lower performance than the standard C++ implementation, which was unexpected given VOLK's intended optimization advantages.

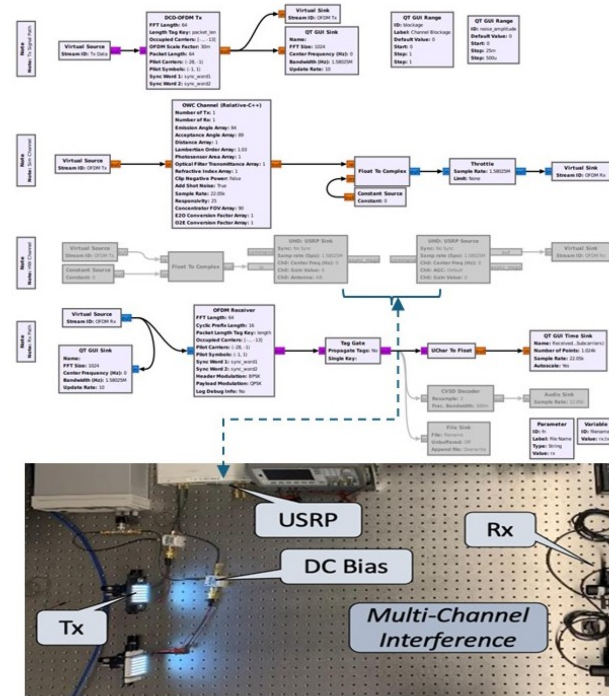


Figure 2: “Digital Twin” model where the OWC channel simulation (left) is run alongside the experimental OWC setup (right)-reproduced from the paper in the GNU Radio conference 2023.

To address this limitation, the implementation was restructured to enable parallelization across incoming samples. By performing gain and noise computations concurrently over multiple input samples, the revised approach achieves improved utilization of SIMD capabilities. This change enabled greater parallel execution by efficiently utilizing SIMD operations, resulting in an improved peak sample rate and more consistent CPU usage under real-time constraints.

As shown in Figure 3, the Python implementation shows peak CPU utilization at relatively low sample rates. The standard C++ implementation demonstrates superior performance; however, the sample-level VOLK optimization achieves the highest efficiency across the full range of sample rates. In contrast, the transmitter-level VOLK implementation exhibits higher CPU utilization than the C++ baseline, indicating reduced efficiency of this parallelization strategy for scenarios involving a small number of transmitters.

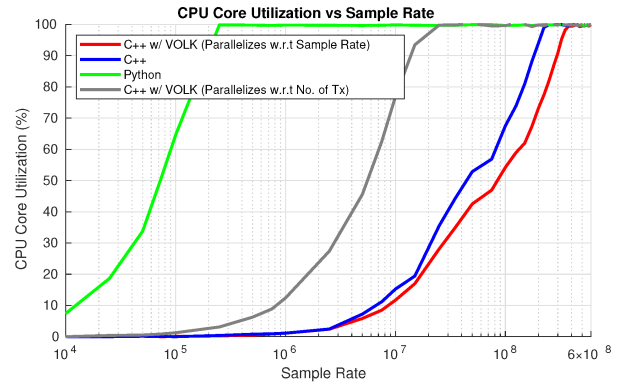


Figure 3: CPU utilization comparison of channel blocks on system 1, implemented in C++ without VOLK, VOLK parallelized at the transmitter level, VOLK optimized at the sample level, and python, for 1 Tx–1 Rx configurations.

4.1.2. Evaluation 2: Hardware Platforms

Figure 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 the highest input rate a block can process before CPU core utilization reaches saturation. Beyond this point, further increases in sample rate do not yield higher throughput due to computational constraints.

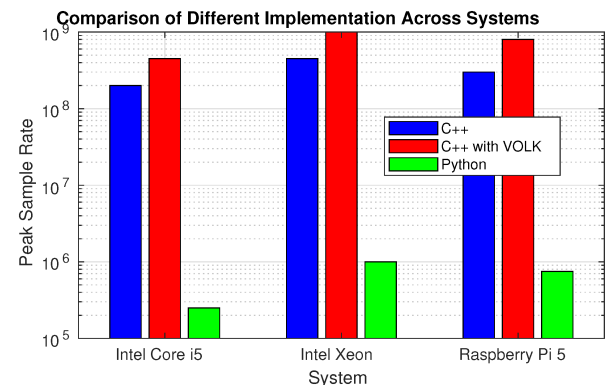


Figure 4: Comparison of different implementations (C++, C++ with VOLK, and Python) of the OWC channel model across different systems, for 1 Tx–1 Rx configurations.

All three systems exhibit distinct hardware characteristics, including CPU architecture and clock speed, number of physical cores, and the number of logical threads per core, among other factors that influence the GNU Radio blocks. In applications employing a thread-per-block scheduler, such as GNU Radio, the number of physical CPU cores, rather than logical threads, has a significant impact on performance, particularly when a higher number of blocks (threads) execute concurrently. In processors supporting simultaneous multi-

threading (SMT), a higher number of threads per core can reduce per-thread performance in GNU Radio's thread-per-block execution model, as block-level threads share the same physical core resources rather than having exclusive access to a dedicated core.

In the Intel Core i5 system, which operates at a maximum clock frequency of 3.20 GHz and supports two threads per core, multiple GNU Radio blocks may be scheduled on logical threads that share the same physical core resources. As a result, each block is unable to exclusively utilize the full execution resources of a physical core, leading to reduce per-thread IPC [15,16] due to resource 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, enabling each GNU Radio block to execute on a dedicated core and more effectively utilize the available clock resources. Additionally, the Intel Core i5 is based on an older microarchitecture, whereas the Raspberry Pi 5's Cortex-A76 cores employ a more modern and energy-efficient design optimized for scalar workloads, which is well aligned with GNU Radio signal-processing workloads.

The Intel Xeon system, equipped with 20 CPU cores, demonstrates superior performance compared to the other two systems. Designed for high-reliability and high-performance computing applications, the Intel Xeon achieves a higher peak sample rate, as illustrated in [Figure 4](#). The results of Evaluation 1 are consistent with these observations, where the C++ implementation with VOLK outperforms standard C++, which in turn outperforms the Python implementation. Similarly, in terms of hardware performance, the Intel Xeon system outperforms the Raspberry Pi 5, which in turn performs better than the Intel Core i5 system. 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 achieved by transitioning from Python to C++ are substantially more significant than the performance differences attributable to the choice of compute hardware.

4.1.3. Evaluation 3: *gr-owc* Code Enhancements

In this work, the OWC channel model implementations are enhanced in two key aspects: (1) core algorithmic optimizations that eliminate unnecessary computations, and (2) new features that extend the functionality of the channel blocks toward 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, features for signal clipping and signal-dependent shot noise are introduced to enable a more realistic representation of the OWC channel and improve simulation accuracy. The impact of these code enhancements is analyzed in the subsequent performance evaluation, which considers the following cases.

- **Case 1:** Initially, the channel block calculated the channel gain for every input sample. Since the gain calculation involves evaluating parameters such as distance, emission angle, and acceptance angle for each Tx–Rx pair, repeated execution of this operation for every sample introduced substantial computational overhead.
- **Case 2:** The implementation above (Case 1) was optimized to precompute the channel gain upon initialization, or whenever a block parameter is modified. Since the gain is independent of the input data or sample, precomputed values are directly reused during processing of each input sample, eliminating the need for repeated gain recalculation. 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 operations.

[Figure 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. Further, 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 introduced features are not applicable in all scenarios (e.g., operation within the LED linear region or conditions where thermal noise dominates and shot noise is negligible), configuration flags have been introduced to enable optional activation of this func-

tionality. 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 introduces additional computational overhead when enabled. Figure 6 illustrates how CPU utilization varies with sample rate under different configurations of these flags. When clip-neg and shot noise addition are disabled (i.e., Case 2 in Figure 5), the channel block requires the lowest computational resources, as no additional processing operations are performed. When only *clip-neg* is enabled, no substantial increase in CPU utilization is observed. 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 first determines the average received power for each receiver. This value is subsequently used to statistically generate and add shot noise to each output sample, introducing additional per-sample computations and thereby increasing CPU utilization. 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.

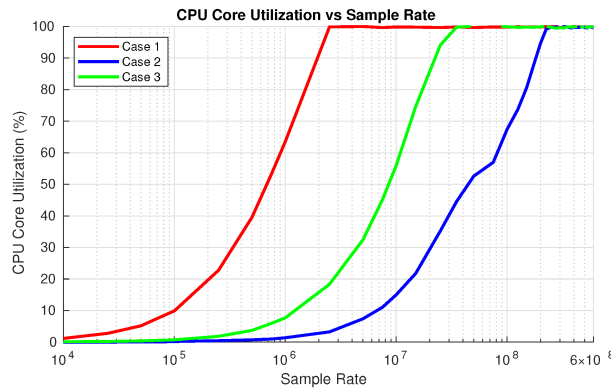


Figure 5: CPU utilization with varying sample rates for cases 1–3, measured on system 1 with C++ implementation and a 1 Tx–1 Rx configuration.

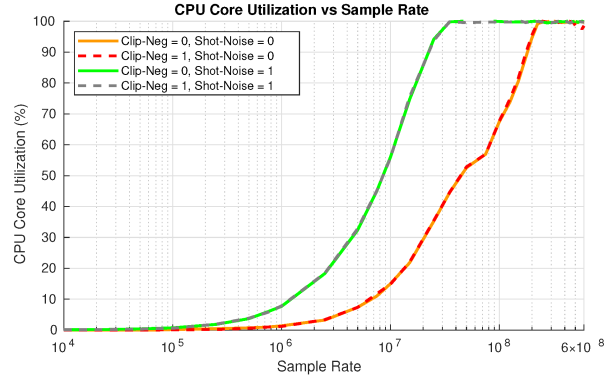


Figure 6: CPU utilization with varying sample rates based on *clip-neg* and *add shot noise* flag states, for 1 Tx–1 Rx configurations.

These results indicate that enabling optional features such as clipping and shot noise introduces additional computational load; however, the overhead associated with evaluating the corresponding configuration switches remains negligible. This justifies the design choice to implement these functionalities within a single configurable block rather than decomposing them into multiple specialized blocks. This approach simplifies flowgraph design and enhances usability, as researchers can easily enable or disable optional features without modifying the underlying flowgraph structure. Moreover, it provides the flexibility to deactivate computationally intensive features when higher sample-rate requirements must be met.

4.1.5. Evaluation 5: Scaling for Multiple Tx/Rxs

The computational complexity of the OWC channel model increases with the number of transmitters and receivers due to the additional signal-processing operations required. The results in Figure 7 illustrate how CPU utilization varies across 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 computation, and all signals are subsequently summed at the receiver, thereby 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.

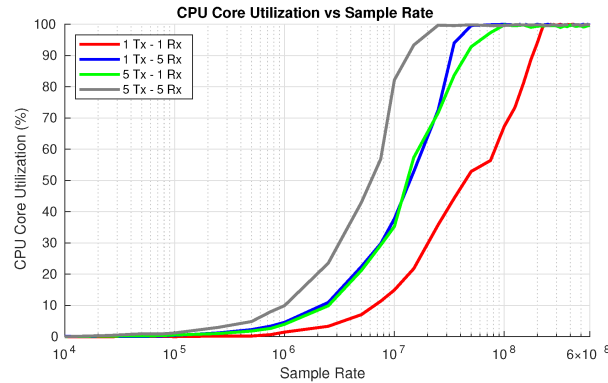


Figure 7: CPU utilization variation across different OWC channel-model Tx-Rx configurations.

4.2. OWC Mod/Demod Blocks

Following the same analysis methodology used for evaluating the channel modeling blocks, a set of modulator and demodulator blocks from *gr-owc* is considered. 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 performs averaging of received samples over each symbol interval and compares the resulting amplitude against a predefined set of decision 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 identifies the pulse location within each symbol interval and decodes the transmitted symbol based on the detected 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 detects the pulse position to recover the transmitted bit and accounts for pulse width to preserve the intended pulse characteristics.

4.2.1. Evaluation I: OOK Modulator

By generating multiple samples per symbol (SPS), the OOK Modulator effectively acts as an interpolator block, while the OOK demodulator functions as 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 results in a higher required sample rate. This also increases the computational load, since more samples must be processed per symbol. These two scenarios can be compared using the different test flowgraphs illustrated in Figure 1. The following tests analyze the effect of SPS and implementation across multiple systems.

The first baseline test of the OOK block analyzes the impact of SPS settings on CPU utilization for the OOK modulator implemented in C++, evaluated on Intel Core i5 and Raspberry Pi 5 systems, as the incoming symbol rate increases. The SPS values tested were 1, 5, and 20.

As shown in Figure 8, increasing the SPS consistently increases CPU utilization, because more samples must be processed per symbol period. For SPS values greater than one, the block is required to process a higher number of samples per second due to the increased output sample count. This results in a higher computational workload even when the input rate remains unchanged. Because a greater number of output samples are generated, GNU Radio schedules the block more frequently to satisfy downstream processing requirements, thereby increasing scheduling overhead (more samples = more memory writes/reads).

Next, CPU utilization of the OOK modulator implemented in C++ is evaluated across three different systems. All systems were tested with SPS = 20 to ensure a consistent load condition. As observed in Figure 9, the Intel Xeon system initially exhibits higher computational resource utilization; however, beyond a certain symbol-rate range, the rate of increase in CPU usage diminishes, after which improved performance is observed. 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 performs better than Intel Core i5, saturating at a higher symbol rate, indicating better efficiency and CPU scaling. The Intel Core i5 system exhibits moderate performance but reaches saturation earlier compared to the Raspberry 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.

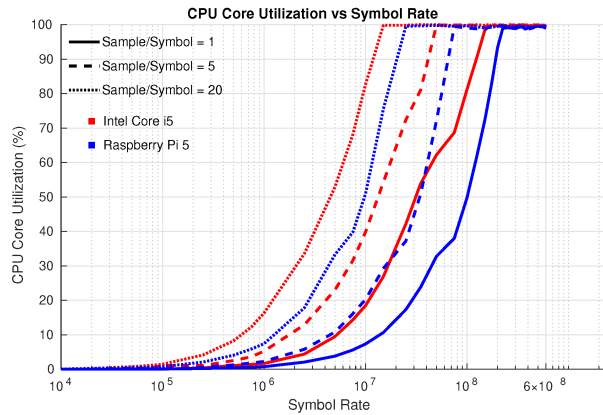


Figure 8: CPU core utilization of OOK modulator (C++) across SPS = 1, 5, 20 on Intel Core i5 and Raspberry Pi 5.

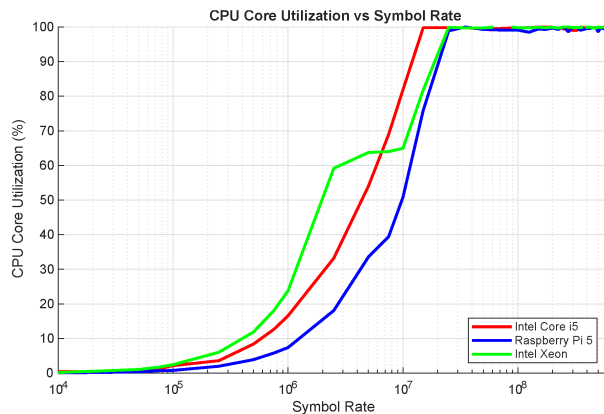


Figure 9: OOK modulator-C++ implementation across Intel Xeon, Intel Core i5, and Raspberry Pi 5 (SPS = 20).

4.2.2. Evaluation 2: OOK Demodulator

The OOK Demodulator block is the counterpart to the OOK modulator block tested in Evaluation 1. The effect of SIMD optimization on OOK demodulator performance is evaluated across different SPS configurations and implementation types, including Python, standard C++, and C++ with VOLK. The C++ with VOLK implementation was specifically optimized to parallelize operations across samples.

As illustrated in Figure 10, which was evaluated using flowgraph V2 (Figure 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.

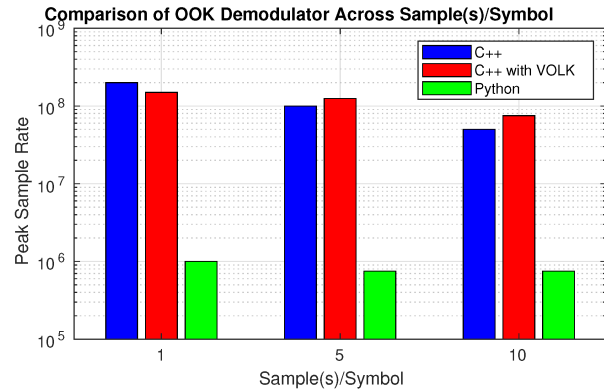


Figure 10: Comparison of OOK demodulator implementations (C++, C++ with VOLK, Python) across different SPS values.

4.2.3. Evaluation 3: PAM Modulator

For PAM implementations, the modulation order (M) relates the sample rate and bit rate in a manner analogous to how SPS relates the 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, both Python and C++ implementations are tested 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 Figure 11.

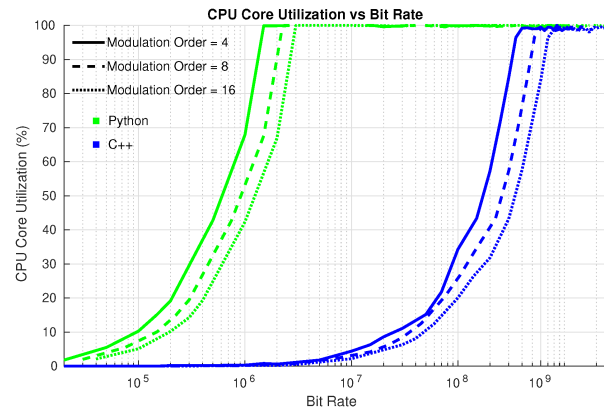


Figure 11: PAM modulator with modulation orders 4, 8, and 16 across C++ and Python implementations.

As the modulation order increases, the number of bits per symbol also increases. This enables the block to maintain the same bit rate while transmitting fewer symbols per second. Consequently, a fixed SPS setting across different modulation orders results in a lower required sample rate at 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* demodulators (OOK, PAM, PPM, and VPPM) were evaluated using the V1 test (Figure 1). All demodulators were tested 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. SPS = 8 configurations of OOK and VPPM were additionally evaluated, operating at the same sample rate and bit rate as the SPS = 32 PAM and PPM configurations with $M=16$, while utilizing 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 Figure 12, the observed trends can be directly attributed to implementation-level design choices. In the OOK modulator, each output sample is generated dynamically from the incoming symbol stream, necessitating per-sample decision logic and repeated value assignments. In contrast, the VPPM implementation pre-computes complete samples-per-symbol waveforms and transfers them to the output buffer using memory copy operations. This approach significantly reduces per-sample computation, which explains the lower CPU utilization observed for VPPM compared with OOK at comparable sample rates. As future work, the OOK modulator could be refactored to adopt a similar pre-initialized waveform (array) and memory-copy strategy, which is expected to further reduce CPU overhead and improve performance.

An important observation from Figure 12 is that, when CPU utilization is considered solely as a function of sample rate, lower-order modulation schemes such as OOK and VPPM appear more efficient, exhibiting lower CPU utilization 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 appar-

ent efficiency of OOK and VPPM when normalized by delivered bit rate. When evaluated at equivalent bit rates, higher-order modulation schemes such as PAM and PPM demonstrate improved CPU utilization efficiency. This trend is also consistent with the results shown in Figure 11, where higher-order modulation schemes perform better when evaluated based on achievable bit rate rather than sample rate alone.

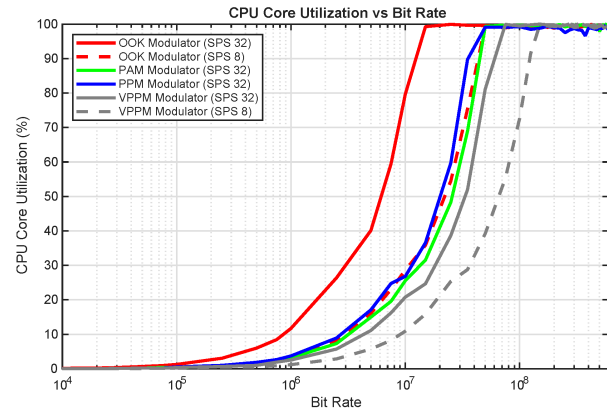


Figure 12: CPU utilization comparison across all modulation schemes (OOK, PAM, PPM, VPPM) on Raspberry Pi 5.

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 is compared against all possible amplitude levels to determine the corresponding waveform, which is then copied to the output buffer. This additional symbol-level comparison and indexing overhead leads to 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 write operations, resulting in 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 (Figure 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 Figure 13, the OOK demodulator exhibits the lowest CPU utilization among all evaluated demodulators. OOK demodulation involves averaging the incoming samples over each symbol interval, followed by a single threshold comparison to distinguish between the two possible transmitted symbols (ON or OFF). This simple processing pipeline results in minimal per-symbol computational overhead.

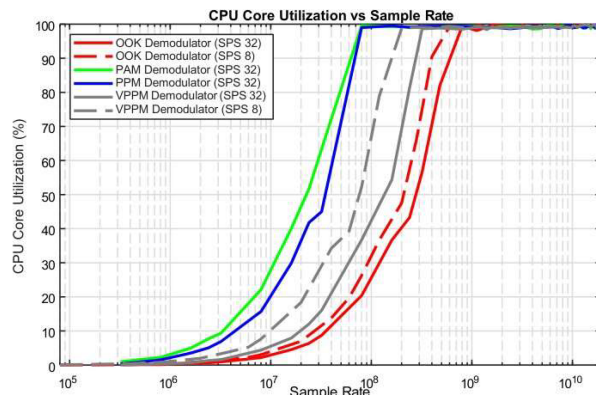


Figure 13: CPU utilization comparison across all demodulation schemes (OOK, PAM, PPM, VPPM) on Raspberry Pi 5.

The VPPM demodulator exhibits higher CPU utilization than OOK but remains significantly more efficient than PAM and PPM. VPPM demodulation employs 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 corresponding 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

results in 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 requires computing correlation metrics for 16 candidate pulse positions and selecting the most likely symbol using a maximum-likelihood decision 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; however, the resulting 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 compared with the other demodulators.

4.3. OWC System (Complete Flowgraph)

The complete OWC system flowgraph shown in Figure 14 represents a baseline configuration for end-to-end performance evaluation of *gr-owc*. The results in Figure 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 the sample rate increases, the demodulator is the first block to reach maximum CPU capacity. Consequently, the overall flowgraph (black curve) also plateaus at this 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 constrained by the processing capacity of the demodulator in the complete system flowgraph.

The overhead in the graph (gray curve) represents the combined CPU usage of the Constant Source, Throttle, and Null Sink blocks, which perform minimal computational operations. In Figure 15, the complete flowgraph reaches $\approx 280\%$ CPU utilization, meaning it effectively consumes ≈ 2.8 CPU cores, indicating its multi-core usage. This highlights a key insight in system-level OWC SDR design: even when upstream blocks are capable of

supporting higher throughput, the block with the highest computational load (here, the demodulator) constrains overall system performance. Identifying and optimizing

such bottlenecks is essential for high-throughput real-time processing in SDR systems.

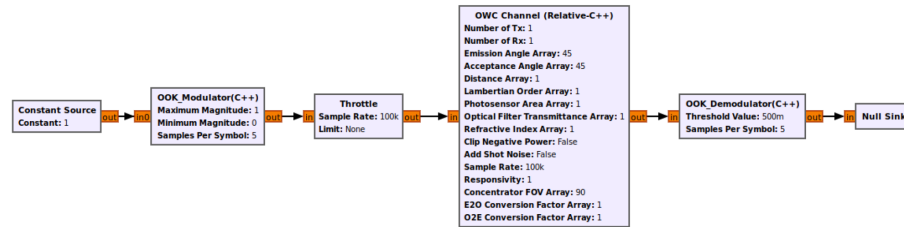


Figure 14: OWC flowgraph used in complete system evaluation.

The surface plot in Figure 16 illustrates 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 through a shared OWC Channel Model, which handles signal transmission across all Tx–Rx pairs.

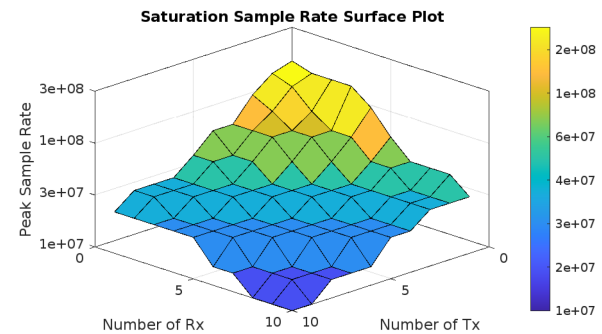


Figure 16: CPU utilization of the complete flowgraph under different transmitter (Tx) and receiver (Rx) configurations, implemented using C++ blocks.

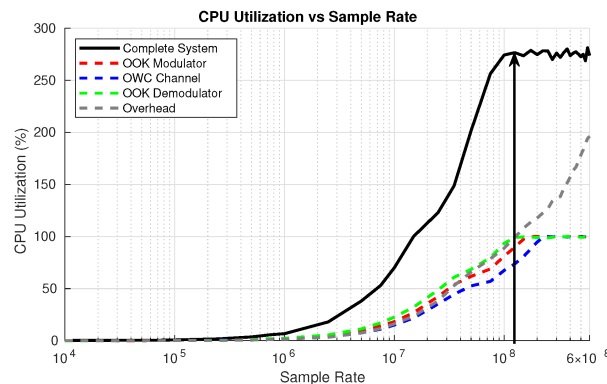


Figure 15: CPU utilization of individual blocks and complete system flowgraph tested on Intel Core i5 using C++ blocks.

Each point on the surface plot represents the saturation sample rate at which the system reaches maximum 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 decrease in the peak sample rate for larger Tx–Rx configurations. The plot highlights the scalability limitations of the system, demonstrating that increasing the number of nodes leads to higher computational demand.

5. Conclusions

This work presents a comprehensive performance characterization of the *gr-owc*, GNU Radio OOT module for optical wireless communication systems, focusing on real-time processing capabilities across diverse hardware configurations and the performance impacts of different implementation strategies. The results provide useful insights for researchers developing custom GNU Radio blocks and modules. Through systematic evaluation of individual signal processing blocks and complete system flowgraphs, CPU utilization trends are analyzed with respect to key operational parameters, including sample rate, modulation order, and samples per symbol.

The results demonstrate that C++ implementations consistently outperform Python in computational efficiency, with additional gains from SIMD-based optimization via VOLK, particularly under high sample-rate workloads. The superior performance of C++ is attributed to its compiled nature, which enables faster execution compared with Python, an interpreted language. This speed advantage is critical in SDR applications, where real-time

operation requires continuous processing of high-rate incoming sample streams.

Channel model analysis revealed that configuration factors such as the number of transmitters and receivers significantly affect CPU utilization, and that precomputing static channel parameters effectively reduces computational overhead. Cross-platform benchmarking further highlights the impact of hardware characteristics—particularly clock speed, memory access, and core count—in 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 upstream or downstream blocks from operating at higher sample rates, thereby effectively limiting overall system throughput. Identifying and optimizing such limiting blocks is essential for achieving scalable real-time performance in SDR systems.

Overall, the study demonstrates that *gr-owc* is a practical and flexible research platform for real-time optical wireless experimentation, making it well-suited for the development of SDR-based OWC systems in multi-user and multi-cell configurations. In dense multi-node environments, experimental proof-of-concept testing is essential for characterizing the performance of novel protocols and signal processing chains. The tools within *gr-owc* can be combined with the broader functionality of GNU Radio to support a simulation-to-experimentation workflow for OWC network testing. Accordingly, the low-level blocks can be extended into system-level models for VLC networks, VLP systems, and hybrid radio-optical networks, enabling practical evaluation across diverse environments and providing real-world validation of system performance metrics. The performance analysis presented in this work demonstrates the limitations of individual blocks while also providing a structured framework for evaluating the performance of more complex flowgraphs. As system complexity increases to support enhanced functionality, identifying bottlenecks within the signal chain becomes essential for determining where optimization efforts should be focused to improve end-to-end system performance.

List of Abbreviations

IPC	Instructions Per Cycle
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 Optimized Library of Kernels
VPPM	Variable Pulse Position Modulation

Author Contributions

Conceptualization, funding acquisition, project administration, resources, supervision: M.B.R.; Data curation, software, investigation, visualization, writing—original draft: K.P.S.; Methodology, validation, formal analysis, writing—review & editing: M.B.R. and K.P.S. All authors have read and agreed to the published version of the manuscript.

Availability of Data and Materials

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

Conflicts of Interest

The authors declare no conflicts of interest.

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.

Acknowledgments

The authors gratefully acknowledge the support provided by the University of Massachusetts Boston and the contributions of the National Science Foundation (NSF) ERI Program toward facilitating this research.

AI Declaration

ChatGPT was used in a limited capacity to support editorial refinement of select text passages. The intellectual framework, 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 the requirements.

References

- [1] I. Abdalla, M. Rahaim, and T. Little, "Interference in multiuser optical wireless communications systems," *Phil. Trans. R. Soc. A: Math. Phys. Eng. Sci.*, vol. 378, no. 2169, Mar. 2020, Art. no. 20190190. [[CrossRef](#)]
- [2] S. Al-Ahmadi, O. Maraqa, M. Uysal, and S. M. Sait, "Multi-user visible light communications: State-of-the-art and future directions," *IEEE Access*, vol. 6, pp. 70555–70571, 2018. [[CrossRef](#)]
- [3] M. S. Demir, S. M. Sait, and M. Uysal, "Unified resource allocation and mobility management technique using particle swarm optimization for VLC networks," *IEEE Photon. J.*, vol. 10, no. 6, pp. 1–9, 2018. [[CrossRef](#)]
- [4] M. Obeed, A. M. Salhab, M. S. Alouini, and S. A. Zummo, "On optimizing VLC networks for downlink multi-user transmission: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2947–2976, 2019. [[CrossRef](#)]
- [5] L. Yang, Q. Zhang, W. Zhang, L. Deng, and H. Yang, "Resource allocation for hybrid visible light communications (VLC)-WiFi networks," *IEEE Access*, vol. 8, pp. 176588–176597, 2020. [[CrossRef](#)]
- [6] E. Blossom, "GNU radio: Tools for exploring the radio frequency spectrum," *Linux J.*, vol. 2004, no. 122, Jun. 2004, Art. no. 4. [[View Online](#)]
- [7] A. Ahmed and M. B. Rahaim, "*gr-owc*: An open source GNURadio-based toolkit for optical wireless communications," in *Proc. 17th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Bologna, Italy, Oct. 11–13, 2021, pp. 48–53. [[CrossRef](#)]
- [8] A. Ahmed et al., "SDR beyond radio: An OOT GNU radio library for simulation and deployment of multi-cell / multi-user optical wireless communication systems," *Proc. GNU Radio Conf.*, vol. 7, no. 1, 2022. [[View Online](#)]
- [9] C. C. Onwuchekwa, M. AminuMukhtar, L. Martinez, A. D. Lemus, V. Planchart, F. Y. Semash, K. Kerby-Patel, H. Zhang, and M. B. Rahaim, "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," *Proc. GNU Radio Conf.*, vol. 8, no. 1, 2023. [[View Online](#)]
- [10] M. Rahaim, A. Ahmed, M. Kulhandjian, and H. Kulhandjian, "Optical OFDMA for multi-cell/multi-user indoor optical wireless networks," in *Proc. 58th Asilomar Conf. Signals, Syst., Comput.*, Pacific Grove, CA, USA, Oct. 27–30, 2024, pp. 742–747. [[CrossRef](#)]
- [11] D. Orhan, L. L. Pilla, D. Barthou, A. Cassagne, O. Aumage, R. Tajan, C. Jégo, and C. Leroux, "Optimal scheduling algorithms for software-defined radio pipelined and replicated task chains on multicore architectures," *J. Parallel Distrib. Comput.*, vol. 204, 2025, Art. no. 105106. [[CrossRef](#)]
- [12] R. H. AlShekh, S. A. Dawwd, and F. N. Qassabbashi, "Comparative review of multicore architectures: Intel, AMD, and ARM in the modern computing era," *Chips*, vol. 4, no. 4, 2025, Art. no. 44. [[CrossRef](#)]
- [13] V. Stoico, A. C. Dragomir, and P. Lago, "An empirical study on the performance and energy usage of compiled Python code," in *Proc. 29th Int. Conf. Eval. Assess. Softw. Eng. (EASE '25)*, 2025, pp. 46–56. [[CrossRef](#)]
- [14] C. Laskos, A. Zubow, and F. Dressler, "Latency analysis of SDR-based experimental C-RAN / O-RAN systems," in *2025 IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, Montreal, QC, Canada, Jun. 8–12, 2025, pp. 893–898. [[CrossRef](#)]
- [15] P. Abad, P. Prieto, V. Puente, and J. A. Gregorio, "Efficient server consolidation through a balanced mix of transformer-based and conventional applications," in *Proc. 39th ACM Int. Conf. Supercomput. (ICS '25)*, 2025, pp. 764–775. [[CrossRef](#)]
- [16] S. Sari, U. Nezir, O. Demir, and G. Kucuk, "Breaking the complexity barrier: Enhancing quality of service in simultaneous multithreading processors," *Aray*, vol. 28, 2025, Art. no. 100602. [[CrossRef](#)]
- [17] A. Ahmed and M. Rahaim, "*gr-owc*: An open source GNURadio-based toolkit for optical wireless communications," *Zenodo*, 2026. [[CrossRef](#)]

Disclaimer/Publisher's Note: The views expressed in this article are those of the author(s) and do not necessarily reflect the views of the publisher or editors. The publisher and editors assume no responsibility for any injury or damage resulting from the use of information contained herein.