2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Improving Flexibility in Hardware-Software Co-Development with Remote Virtual Prototypes

Przemysław Mikluszka (*Przemyslaw.Mikluszka@imgtec.com*)
Patryk Górniak (*Patryk.Gorniak@imgtec.com*)
Imagination Technologies, Wrocław, Poland

***Abstract*— This paper proposes a method for remote integration of GPU Virtual Prototypes (VPs) into QEMU-based full-system simulations, aimed at improving flexibility and reducing setup overhead in hardware-software co-development workflows. The solution employs a client-server architecture layered over a proprietary API to enable seamless switching between VP implementations. While local testing showed minimal performance degradation, real-world scenarios with moderate network latency revealed significant overhead due to high-frequency memory transactions. To mitigate this, optimizations such as server-side memory management, deferred batched writing, and dynamically sized chunked reading were introduced. These enhancements improved performance for compute-bound workloads, though limitations remain for interactive applications.**

Keywords*— *hardware-software co-development, Virtual Prototypes, full-system simulation, QEMU, remote execution

I. INTRODUCTION

The increasing competitiveness of the industry and the pressure to reduce time-to-market have made hardware-software co-development a key aspect of modern engineering workflows. A common practice is the use of Virtual Prototypes (VPs) - simulation models that replicate hardware functionality at a chosen level of abstraction. This concept extends to full-system simulation through system virtualizers such as QEMU or Gem5, enabling interaction with VPs as if they were physical devices, typically within a Linux-based environment.

From a software development perspective, this approach provides a practical and efficient means for early testing and validation. It aligns with the growing trend of leveraging virtual prototyping as a core methodology for developing drivers and application software prior to the availability of the final IP design [1]. As the IP design matures, these models can be refined to increase accuracy, enabling more thorough validation. This is particularly important for debugging device drivers, which play a critical role in operating system stability. Faulty drivers can lead to significant issues post-deployment, making early detection and resolution both technically and economically essential. Moreover, as VPs become more accurate, they can serve not only as debugging tools but also as platforms for analyzing the performance and resource utilization of the designed IP under performance-critical workloads.

As the number of available models continues to grow, a significant challenge arises in integrating each of them into a full-system virtualizer. One effective approach is to design these models to conform to a unified API, which serves as a standardized communication interface. This ensures that the virtualizer's communication layer remains consistent, regardless of which specific model is being used. As long as VPs remain purely software-based, they can typically be swapped out with relative ease throughout various IP development phases, allowing for the gradual introduction of more accurate models. However, when more advanced modeling techniques are employed—such as hardware emulation using platforms like Cadence Palladium—integration becomes considerably more complex. These setups often require substantial configuration effort, which must be repeated each time a new developer needs access.

To address this, minimizing the frequency of such setups becomes a clear objective. Consequently, the potential for remote access to these models was explored, aiming to streamline development workflows and reduce setup overhead. This paper presents a method for remotely utilizing VPs within full-system virtualizers, enabling seamless transitions between different models of the same hardware design. By adopting this approach, the overhead typically associated with reconfiguring environments when switching between models can be

significantly reduced or eliminated. Furthermore, it allows for the application of a consistent validation methodology across all IP model variants, regardless of their level of abstraction or implementation platform.

II. APPLICATION

The solution has been developed primarily for a QEMU-based simulation environment, which provides a Linux system equipped with our software components (e.g., drivers) and one or more GPU Virtual Prototypes (VPs). From the perspective of the virtual system, these VPs behave as if they were real hardware devices. Communication between the system and the models is handled through Imagination's proprietary API, which remains consistent across all VP types. This API defines a set of functions that facilitate interaction between the simulator and the rest of the system. Specifically, these functions enable the system to read from and write to device registers, allow the device to request data from device memory (which is modeled and managed outside the simulator), and support the generation of interrupt signals by the device. As a result, the communication model must accommodate bidirectional, stateful interactions. This requirement rules out the use of common remote procedure call mechanisms such as gRPC, which are typically designed for stateless, client-initiated communication.

The initial approach involved introducing an additional communication layer on top of the existing proprietary API. A client-server architecture was adopted, where the QEMU-based system acts as the client, packages the API function calls into Protocol Buffers (Protobuf) messages, and sends them to the server. Upon receiving messages, server unpacks function calls and forwards them to the GPU simulator for processing. Communication is handled over TCP sockets using the ZeroMQ (ZMQ) library. Given that the proprietary API requires bidirectional communication—allowing both the host and the device to initiate function calls—the ZMQ Dealer-Router socket pattern was selected. This pattern supports many-to-many bidirectional communication, although only one-to-one communication has been used during development.

The server and client exchange messages using the request-reply communication pattern; however, as previously mentioned, either side can initiate the exchange. Upon establishing a connection, a session is created, reserving a single VP instance for the exclusive use of the connected client until disconnection. This behavior is repeated for each new client that connects. Since the communication is stateful, a mechanism is required to distinguish between clients. In the ZMQ Dealer-Router pattern, each new dealer (client) is assigned a unique identifier upon sending its first message to the router (server). This identifier is used to differentiate between clients and must be prepended to each reply message to ensure it is routed correctly. The initial design is illustrated in Figure 1.

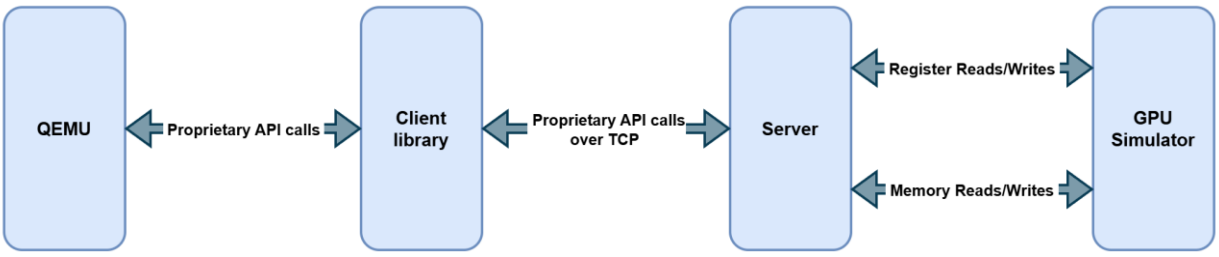

Figure 1. Initial architecture of the solution

Initial testing, with both the server and client running on the same machine, yielded promising results, suggesting that communication overhead was negligible. The observed performance penalty ranged from 281% to 347% of the original execution time, depending on the workload. However, when transitioning to a real-world scenario—where the server and client operated on separate machines with a network latency of approximately 60 milliseconds —the approach became impractical. Simple workloads that typically completed in a matter of seconds began taking over an hour, often without producing any results. These findings highlighted the need for significant optimization, as the original implementation was not viable under realistic network conditions.

Upon further analysis, it was identified that the primary performance bottleneck stemmed from the use of overly narrow data containers within the API's data transfer mechanisms. As a result, even during GPU firmware initialization, the system generated tens of thousands of individual requests. While this produced negligible overhead in local (localhost) testing, it led to severe performance degradation in real-world scenarios involving separate machines and a moderate network latency. In such conditions, each request incurs a minimum delay equal to the round-trip latency, meaning that the cumulative effect of these high-frequency, small-payload transactions results in infeasible execution times.

To make the remote execution approach viable, it was concluded that the server must independently manage the device memory. To support this, a mechanism available in QEMU was leveraged, which allows interception of all memory transactions directed to device memory issued by the driver. This enabled direct forwarding of memory writes from the driver to the server, allowing it to maintain and manage the memory state. However, this solution only partially addressed the problem. The mechanism initially supported transfers of up to 8 bytes per transaction, which inadvertently increased the number of memory operations and, consequently, the communication overhead. This exacerbated the performance issues rather than resolving them.

Further analysis revealed that device memory write requests could be deferred and sent in batches, triggered by the initiation of another memory transaction by the host. This insight led to a significant reduction in the number of network transactions, resulting in improved execution times. Nevertheless, the overall volume of requests remained too high for practical use.

Inspired by data synchronization techniques used in large file transfers—particularly, the Delta Transfer Algorithm employed by tools like rsync [2]—a more efficient strategy was introduced. The core idea involves partitioning data into larger blocks or chunks and transferring them as a whole, rather than sending small fragments individually. This concept was applied to device memory read operations: upon receiving a read request, the server responds with a substantially larger memory chunk containing the requested data. If the host subsequently accesses data within that chunk, it can do so locally without issuing additional memory requests. However, the chunk is invalidated as soon as the host attempts any memory operation other than a read. This optimization significantly reduced the total number of memory transactions required during execution.

Lastly, it was observed that a common execution pattern involves the driver writing data to device memory, followed by validation of specific portions of that data. These memory accesses often span regions larger than the default chunk size. To address this, a mechanism was introduced to track contiguous memory regions written in a single batch. When a subsequent memory read request originates from the same starting address, a chunk of the previous write size is returned. In all other cases, a default chunk size—configurable via a command-line parameter—is used. This technique, referred to as dynamic chunk sizing, further optimizes memory access patterns by adapting chunk sizes to match recent write activity. The changes in the number of memory transactions required to execute a simple test application are presented in Table 1. The final system design is illustrated in Figure 2.

Table 1. Changes in the number of memory transactions during optimisation process

Approach version	Number of transactions	Portion of baseline
Initial (baseline)	95409	100 %
Server-side memory	937876	983 %
Batched writes	126448	132 %
Chunked reads	2430	2.54 %
Dynamic chunk sizing	2233	2.34 %

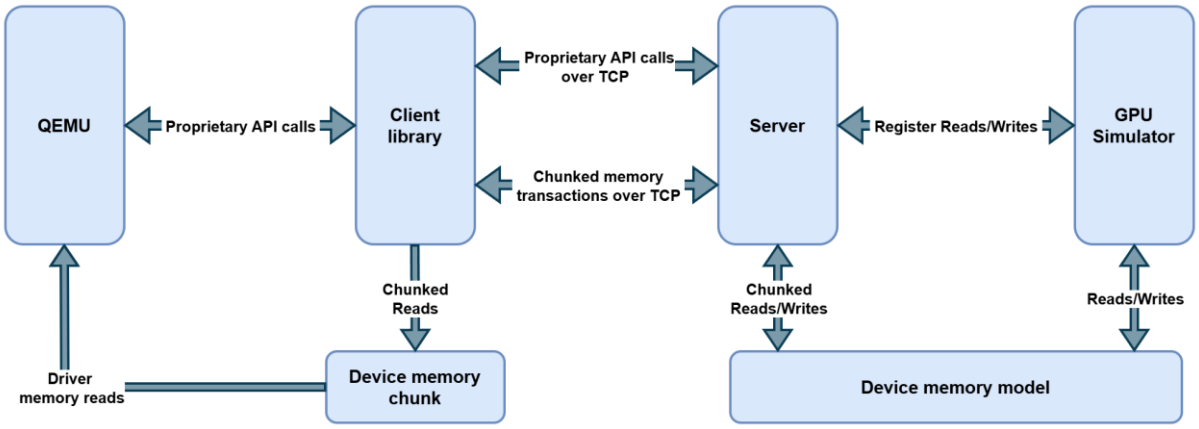

Figure 2. Final architecture of the solution

III. RESULTS

The final solution was evaluated within a QEMU-based environment, with the server and client operating on the same local network and a measured latency of approximately 60 milliseconds between the two machines. A purely software-based functional GPU model was used as the underlying simulator.

Three types of applications were selected for testing:

- A simple 3D application utilizing the OpenGL API (referred to as **OGL**).
- A simple 3D application utilizing the OpenGL ES API, including shader compilation (referred to as **OGLES**).
- A compute application utilizing the OpenCL API to perform FFT calculations on a large image (referred to as **IMGFFT**).

For the 3D applications, execution time was measured for rendering 1, 10, and 50 frames. Additionally, the OGL application was tested for rendering a single frame both with and without GPU firmware initialization, to assess the overhead introduced by that operation. The results of these experiments are presented in Table 2.

Table 2. Execution times for experiments using 3 types of workloads

Application	**Non-remote baseline**	**Remote run with min. chunk size of**			
		512 bits	**2048 bits**	***4096 bits***	***8192 bits***
OGL 1 frame (cold start)	3 s	236 s	165 s	151 s	159 s
OGL 1 frame	2 s	96 s	96 s	94 s	95 s
OGL 10 frames	21 s	125 s	123 s	122 s	125 s
OGL 50 frames	103 s	253 s	243 s	239 s	430 s
OGLES 1 frame	4 s	512 s	515 s	506 s	509 s
OGLES 10 frames	21 s	539 s	536 s	530 s	533 s
OGLES 50 frames	103 s	683 s	661 s	649 s	659 s
IMGFFT	296 s	424 s	377 s	355 s	392 s

The first key observation is the existence of an optimal minimum chunk size when using dynamic chunk sizing. This parameter must still be defined for cases where memory read requests target addresses that were not previously recorded during write operations. The collected data supports the initial intuition: setting the chunk size too low

results in a higher number of memory requests, which increases overall execution time. Conversely, setting it too high leads to the transmission of large data blocks, which can saturate network bandwidth—especially when much of the transmitted data is never actually accessed. These findings suggest that selecting a moderate, balanced chunk size is likely to yield the best performance, minimizing both the number of transactions and unnecessary data transfer. In this case, the most favorable results were observed with a minimum chunk size of 4096 bytes.

The second observation is that the compute workload exhibited proportionally lower overhead compared to the 3D rendering workloads. A similar trend was observed in tests where multiple frames were rendered. This indicates that the proposed solution is better suited for compute-bound workloads or scenarios where minimal additional data is written between successive frame renders.

Additional experiments were conducted to evaluate the impact of network latency on execution time. To simulate varying conditions, similar tests were repeated while artificially increasing latency using Linux kernel traffic control mechanisms. The experiments were rerun with latencies set to 90 milliseconds and 120 milliseconds, while maintaining a minimum chunk size of 4096 bytes. The observed increase in network overhead—measured as the difference in execution time between using the remote mechanism on different machines and running it locally—appears to be linearly dependent on network latency. A similar analysis was performed to examine the relationship between overhead and the number of memory transactions, which also confirmed a linear dependency. These trends are visualized in Figure 3.

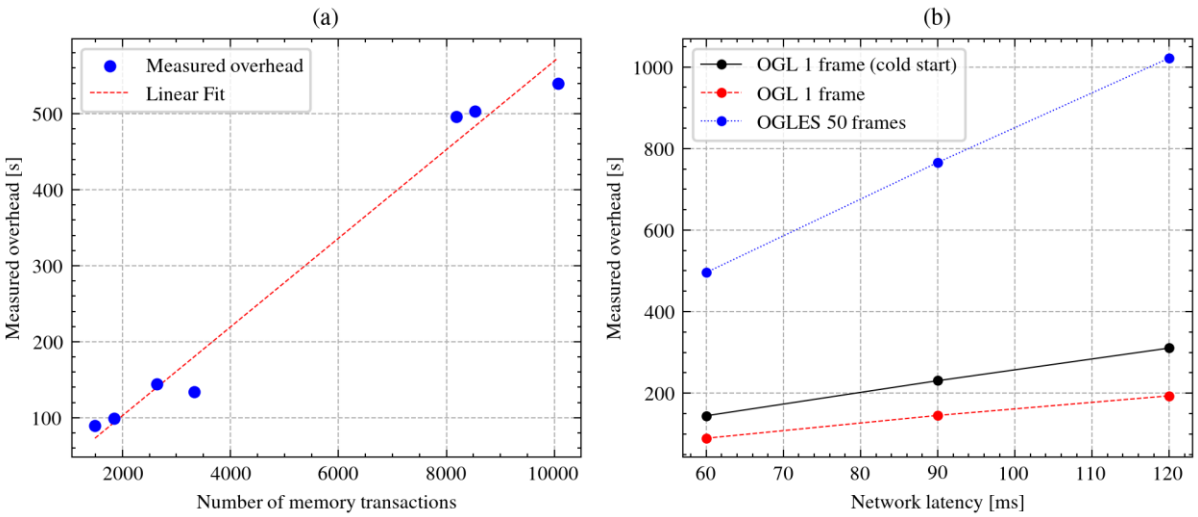

Figure 3. Analysis of network overhead as a function of (a) the number of memory transactions and (b) network latency

In its current form, the solution shows potential, though it may require further refinement to fully support interactive or graphics-based applications. The most promising use case appears to be compute-bound workloads, where the relative performance overhead is significantly lower. In contrast, for simpler applications—such as those typically used during debugging—the solution proves less effective due to the high volume of memory transactions required.

The primary performance bottleneck was identified as the frequency and volume of memory synchronization operations between the client and server. Addressing this limitation will be the focus of future work, with an emphasis on developing more efficient mechanisms for synchronizing device memory.

A key challenge moving forward will be maintaining compatibility with the proprietary API, as ease of integration remains one of the core advantages of this approach. Balancing performance improvements with API conformance will be essential to enhancing the solution's viability in real-world scenarios.

IV. CONCLUSION

This paper presented a method for enabling remote access to GPU Virtual Prototypes within a QEMU-based simulation environment, aiming to reduce setup overhead and improve flexibility in hardware-software co-development workflows. By introducing a client-server architecture layered over a proprietary API, the solution allows seamless switching between different VP implementations without modifying the surrounding system.

Initial results showed that while the approach is functionally correct and performs well in local environments, it suffers from significant performance degradation under realistic network conditions. The primary bottleneck was identified as the high volume of fine-grained memory transactions, which led to excessive communication overhead when latency was introduced. To address this, several optimizations were implemented, including server-side memory management, deferred batching of memory writes, and a dynamic chunk sizing mechanism inspired by delta transfer algorithms. These improvements significantly reduced the number of memory transactions and improved execution times, particularly for compute-bound workloads.

The solution currently encounters limitations when applied to interactive or graphics-heavy applications, primarily due to its sensitivity to network latency. Future work will focus on further reducing synchronization overhead while maintaining compatibility with the proprietary API. Achieving this balance will be key to making the approach viable for a broader range of use cases, including early-stage driver development and performance analysis across diverse VP implementations.

REFERENCES

[1] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without devices. In Proc. of OSDI, 279–292, 2012.

[2] A. Tridgell and P. Mackerras, “The rsync algorithm,” Australian National University, Technical Report TR-CS-96-05, 1996.