Towards Cycle-accurate Simulation of xBGAS

Jie Li*, John D. Leidel†, Brian Page‡ and Yong Chen*
*Texas Tech University, USA
†Tactical Computing Laboratories, USA
‡Laboratory for Physical Sciences, USA
Email: *{jie.li},{yong.chen}@ttu.edu, †jleidel@tactcomplabs.com, ‡bapage@lps.umd.edu

***Abstract*—High-performance computing (HPC) systems are evolving to address big-data and data-intensive workloads, shifting from monolithic architectures to integrated setups with microprocessors, accelerators, and advanced interconnects. However, this transition introduces complexities, latency challenges, and performance bottlenecks in large-scale parallel applications. To tackle these issues, the Extended Base Global Address Space (xBGAS) project enhances memory addressing through innovations in Instruction Set Architecture (ISA) and microarchitecture. Leveraging RISC-V's extensibility, xBGAS integrates an extended register file and new instructions, enabling efficient global memory access. This paper introduces *REV-xBGAS*, a cycle-based simulator using the Structural Simulation Toolkit (SST) to model xBGAS-enabled processors. With SST's modularity, *REV-xBGAS* allows easy configuration of network latencies, bandwidths, and topologies, enabling performance evaluations under varied conditions.**

***Index Terms*—Architecture Simulation, High-performance Computing (HPC), RISC-V, Structural Simulation Toolkit (SST)**

I. Introduction

The growing volume of big-data and data-intensive workloads, exemplified by Large Language Models (LLMs) in high-performance computing (HPC) systems, is necessitating a transformation in computer architectures and programming models. Despite the progress made in fabrication and device packaging, the components of high-performance computing systems are designed independently and require intricate, multilevel software stacks to link the different parts. Latencies that are not desired, complexity, and a decrease in performance of large-scale parallel applications are often the result of this. The Extended Base Global Address Space (xBGAS) project is proposed to extend the addressing capabilities at the Instruction Set Architecture (ISA) and microarchitecture level to reduce the number of software layers and simplify the implementations of the Partitioned Global Address Space (PGAS) programming model [1]–[4]. It leverages the extensible nature of the RISC-V architecture by integrating an extended register file and introducing a set of new instructions to provide global, scalable memory addressing support [2], [5].

It is worth noting that xBGAS cannot run on current commodity microprocessors as the new register file and new instructions are introduced. Previous developments and experiments mainly rely on the functional simulator built on the traditional RISC-V simulator, *Spike*, and use MPI to simulate communication between processing elements (PEs) [2], [6]. Although this approach provides a relatively fast environment for developing and testing the xBGAS runtime library and compiler, it is limited in performance analysis because of its inability to model the time behavior of hardware components. Moreover, it is unable to assess the effects of extended instructions on various networks, which is a major shortcoming when building and evaluating a new distributed system.

In this paper, we present *REV-xBGAS*, a cycle-based simulator that is capable of simulating xBGAS-enabled processors using the Structural Simulation Toolkit (SST) [7], [8]. Taking advantage of the modularity and extensibility of SST, we can configure the xBGAS-enabled distributed system to have different network latencies, bandwidths and topologies, and evaluate the system performance under different configurations. Our contributions are summarized as follows:

- We present the detailed design and implementation of REV-xBGAS, reflecting the microarchitecture-level changes to achieve the extended global address space.
- We configure and run the xBGAS-enabled distributed system with various network topologies, demonstrating its functional correctness.
- We simulate xBGAS in a cycle-based manner, allowing us to gain a deeper understanding of its behavior and performance, and use this knowledge to optimize the runtime library.

This paper is structured as follows. First, we provide the background of xBGAS in Section II. Next, we present the detailed design and implementation of REV-xBGAS in Section III and demonstrate the evaluation of xBGAS using different network configurations in Section IV. Section V discusses future directions and concludes the paper.

II. Background

xBGAS is a novel approach to providing high performance, scalable, shared memory address spaces based on the open-source RISC-V ISA. In this section, we give a brief overview of the xBGAS microarchitecture and instructions.

A. xBGAS Addressing Model

Remote data access in xBGAS is implemented by mapping remote resources into the extended address space, such that inter-PE memory operations can be conducted with memory access semantics (load/store). To provide extended address spaces, xBGAS introduces a set of 32 xBGAS-specific registers to the RISC-V RV64 register file. These extended registers, denoted as *e0-e31*, are analogous to standard general purpose

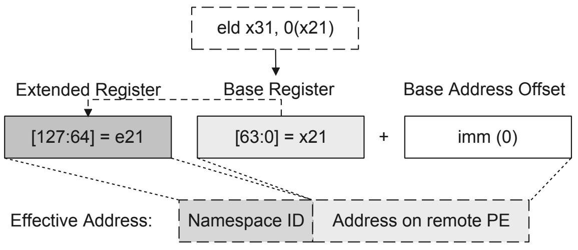

Figure 1. xBGAS Addressing Model

registers (*x0-x21*)and contain the namespace ID that denotes the logical storage location of the remote data object.

Figure 1 gives an example of the `xBGAS` addressing model for the `xBGAS` *eld* instruction, which loads an integer value from a remote PE and stores the value in *x31*. Note that in this instruction, the base register (*x21*) contains a standard 64-bit memory address, and together with the immediate value *0*, they form the target address. The corresponding extended register (*e21*, shares the same index as the base register) contains the logical namespace ID.

B. xBGAS ISA Extension

`xBGAS` introduces *four* sets of extended instructions for remote data operations and value manipulations of the extended registers. These instructions are summarized as follows:

Integer Load/Store Instructions: Similar to the base *load* and *store* instructions of the standard RISC-V ISAs, `xBGAS` introduces *extended* load and store instructions for remote data operations using the RISC-V *I-type* and *S-type* instruction encoding, respectively. The mnemonic of these instructions are simply adding *e* to the base load and store instructions, and the extended register corresponding to the base register is implicitly used for specifying the namespace ID. For example, *eld* is the counterpart of the *ld* instruction; it loads a double-word size of data from remote PE. `xBGAS` supports operations on data with sizes ranging from a *byte* to a *double word*.

Raw Integer Load/Store Instructions: Unlike the previous set of load/store instructions where the extended register is implicitly used, the instructions in this category allow specifying the extended register explicitly. This is achieved through the RISC-V *R-type* encoding, where *three* registers can be encoded in a single instruction. This permits applications to perform more complex operations and reduces the number of instructions for collective operations [4]. An example of *raw* integer load instructions is as follows:

$$erld\ rd, rs1, ext2 \tag{1}$$

It loads a *double-word* (64-bit) value from the effective address formed by combining the namespace ID stored in *ext2* and memory address stored in *rs1*, and saves the value in *rd*.

Bulk Integer Load/Store Instructions: The previous two categories of instructions are only capable of transferring a single data element with up to the register-width (64-bit) size. Transferring large data elements with patterns, which is a common practice in distributed programming, requires a loop of previous `xBGAS` load/store instructions, resulting in a significant increase in the number of instructions. Instructions in this category are designed to improve the transfer of large data elements by encoding the data patterns in one instruction. This allows a single *bulk* load/store instruction to move data elements that are larger than the size of a register. Instead of being stored in registers, the data elements are transferred to memory. The RISC-V *R-4 type* encoding is used to facilitate the encoding of *four* registers. An example of this category of instructions is:

$$ebld\ rd, rs1, rs2, rs3 \tag{2}$$

It loads m elements (encoded in *rs2*) with n strides (encoded in *rs3*) from memory address x (encoded in *rs1*) of namespace y (encoded in the *extended* register of *rs1*), and stores the data elements at the starting address encoded in *rd* with the same pattern as the data source.

Address Management Instructions: The final category of `xBGAS` instructions do not operate directly on remote data; instead, they provide the capability to manipulate the values in the extended register. These instructions follow the standard RISC-V *I-type* instruction encoding and make use of the core RISC-V ALU in the same way as moving data between general-purpose registers. For example, to set *e21* to have a value stored in the general purpose register *x10*, we may issue the following instruction:

$$eaddie\ e21, x10, 0 \tag{3}$$

The comprehensive description of `xBGAS` instructions can be found in the `xBGAS` specification [9].

III. xBGAS Cycle-accurate Simulation

The extended addressing capabilities of `xBGAS` require changes in the processor architecture to bridge the in-system bus architecture with the network fabric. For example, when the namespace ID refers to the data object on PEs on remote compute nodes, the in-system bus packets must be translated to be routed to the correct resource on the fabric. Designing and implementing a cycle-accurate simulator for xBGAS-enabled processors should reflect the architecture changes and model the behaviors of each component. In this section, we describe the details of the implementation of REV-xBGAS.

REV-xBGAS is built on the *REV* RISC-V CPU model [10], which is an open-source project designed to provide cycle-based simulation capabilities of arbitrary RISC-V cores with 5-stage pipelining. REV utilizes the Structural Simulation Toolkit (SST) as the core parallel discrete event simulation framework and can be attached to other SST elements for full system and network simulations. Figure 2 shows the overall architecture, where the customized and new components added to the REV CPU model are highlighted in colors, with some modifications to the CPU register files and decoder. These new components are described below.

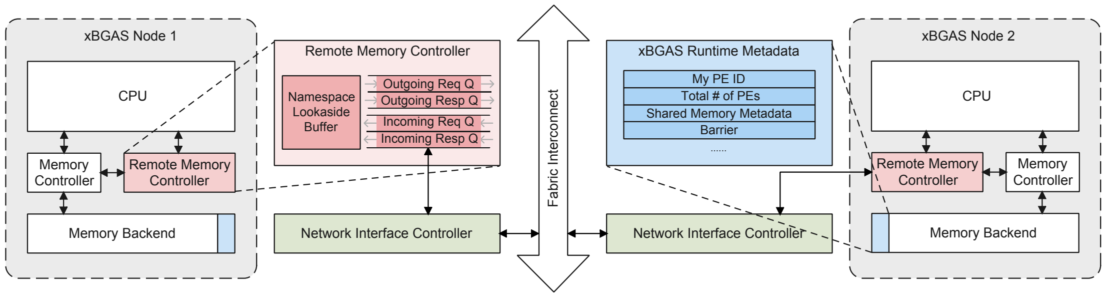

Figure 2. REV-xBGAS Architecture. Colored blocks are the main components added to support xBGAS simulation. The red block represents the remote memory controller, the green block is the customized NIC capable of packetizing information for remote memory operations, and the blue block is the static memory space for the runtime metadata.

A. *xBGAS Extended Registers*

Incorporating the extended registers utilized by `xBGAS` involves straightforwardly creating a register file data structure as defined in REV. This structure is essentially an array composed of 32 unsigned 64-bit integers, and accessing and updating the register values means simply reading and altering the corresponding elements in the array. These extended registers are solely designated for `xBGAS` instructions, ensuring that the existing RISC-V instructions implemented in REV remain unaffected and do not require modifications.

B. *Remote Memory Controller*

The Remote Memory Controller (RMC), colored red in the diagram, is designed analogously to the local memory controller and provides interfaces to access the remote memory address space. There are several data structures implemented in this component, a *Namespace Lookaside Buffer* (NLB) and four *queues* that handle outgoing/incoming requests/responses, as shown in the zoomed-in block.

Similar to the Translate Lookaside Buffer (TLB), which accelerates the translation of a virtual address to a physical address, the NLB takes a namespace ID as input and translates it to a global resource address on the fabric. If the NLB translation result refers to the local node address, the memory request is dispatched to the memory controller and fulfilled by the local memory hierarchy. Otherwise, the memory request is sent to the outgoing request queue which is waiting to be sent to the fabric. NLB is initialized and distributed to each participating node when the simulation is initialized.

RMC handles not only requests originating from the local CPU issuing a `xBGAS` load/store instruction but also those from remote processing elements. Temporary storage of these requests is managed using the *outgoing request queue* and the *incoming request queue*. The outgoing requests, structured as CPU packets, are forwarded to the Network Interface Controller for conversion into fabric packets. On the other hand, the incoming requests, already transformed into CPU packets, are routed to the memory controller to perform operations in the local memory. Responses that fulfill these incoming requests find their place in the *outgoing response queue* and are subsequently routed to the PEs through the NIC. Additionally, incoming responses, comprising memory operation responses sent from remote PEs, are temporarily stored in the *incoming response queue* to facilitate the proceeding of stalled `xBGAS` load/store instructions.

C. *xBGAS NIC*

The `xBGAS` NIC is similar to the traditional NIC but is expanded to be capable of packetizing additional information for remote memory operations. This additional information includes *Src, PktId, Addr, Opcode, Size, Nelem, Stride*, and *Data*.

The source node ID (*Src*) helps determine which node should receive the responses for remote memory operations. The packet ID (*PktId*) serves to match responses with requests, getting assigned a new value when a request arises. Meanwhile, the memory address (*Addr*) specifies where the data object on the remote nodes starts. As for the operation code (*Opcode*), it signifies various operation types like *load*, *store*, *bulk load*, and *bulk store* for remote memory operations. Additionally, the element size (*Size*) designates the size of each data element in bytes. Alongside this, the fabric packet contains information about the number of elements (*Nelem*) and the stride (*Stride*) specifically for bulk transfers. Lastly, the data field (*Data*) takes on the role of carrying the payload of remote memory operations.

D. *xBGAS Instruction Decoding*

Except for these newly introduced simulated hardware components, the modification made to the REV CPU model incorporates the decoding support for `xBGAS` extended instructions. As mentioned above, the `xBGAS` instructions follow the existing encoding formats (*I-, S-, R-, R4-type*) defined in RISC-V ISA and therefore do not require additional logic to extract the “opcode”, “funct” fields, register fields, and immediate bits. The only modification required is to link the combination of “opcode” and “funct” fields with the implementation functions of `xBGAS` extended instructions, following the `xBGAS` specification [9].

It should be noted that the `xBGAS` CPU only loads executables from the local memory hierarchy, and therefore the *fetch* stage of the pipeline does not involve the participation of newly added components. However, when a `xBGAS` instruction is decoded, the extended registers will be used to decode the namespace ID, and the *memory access* and *write back* stages will leverage the new components to perform remote memory operations and destination updates.

E. xBGAS Instruction Execution

The execution of the `xBGAS` instructions is done through a header file that provides the instruction implementations and an encoding table to find the implementation functions after decoding the instructions.

We use the instruction $eld\ x31,\ 0(x21)$ as shown in Figure 1 to exemplify the construction of the implementation function for the *eld* instruction. When this instruction is decoded, the implementation function initially accesses the general-purpose register *x21* and computes the address in the remote processing element (PE), given by $x21 + 0$. Subsequently, the implementation function invokes the API provided by the remote memory controller (RMC) to initiate a remote *load* request, providing the namespace ID (encoded in *e21*) and the calculated address as parameters. The RMC then handles the remote memory operation, and once the corresponding response is received, it updates the destination register (*x31*). The instruction stalls until the destination register has been successfully updated.

The encoding table is constructed as a C++ vector of struct entries, with each entry corresponding to an individual instruction. Within REV, the encoding table holds all enabled instructions. For each unique combination of the ”opcode” and ”funct” fields, there exists a single corresponding implementation function. The REV CPU decodes instructions and, by examining the encoding table, identifies the right implementation function to be executed.

F. xBGAS Runtime Metadata

The `xBGAS` runtime has been designed to offer APIs to developers through a C-based library. This empowers applications to conduct memory allocation and data movement utilizing *gets* and *puts* with the `xBGAS` instruction set extensions. Applications employing the `xBGAS` model are linked to the runtime library, and their compiled executables are run within the REV-xBGAS simulator for simulation.

However, there exists a disconnect between the original `xBGAS` runtime library and the simulator. While the `xBGAS` runtime library supports environment routines that provide essential details (such as PE ID, the count of participating PEs, etc.) to the calling application, this environment information is solely accessible to the simulator itself once the simulation has been initialized. To bridge this gap between the `xBGAS` runtime library and the simulator, we designate a distinct static memory space of 4KB within each simulated `xBGAS` node, visualized as the blue block in Figure 2. This designated memory space is employed as an interface for the exchange of environment information. In addition, the `xBGAS` runtime library also manages a data structure that records the metadata of shared memory information across all participating PEs and a bitmap that is used for synchronization among PEs (i.e., *barrier*). Recognizing the indispensability of this information for all PEs, we utilize the aforementioned static memory space to store these data structures.

TABLE I
CONFIGURATIONS OF THE XBGAS DISTRIBUTED SYSTEM

Subsystem Parameters		**Values**
Nodes	# of Nodes	4, 8, 16, 32, 64, 128, 256
	# of Cores per Node	1
	CPU Architecture	RV64G + xBGAS
	CPU Clock Freq.	2.5 GHz
	Memory cap. per Node	1 GB
Network	Latency	40 μs
	Bandwidth	10 GB/s
	Flit Size	32 Bytes
	Buffer Size	512 Bytes
	Topologies	Star, Torus, Fat tree

IV. Experimental Results

Structural Simulation Toolkit (SST) offers a wide selection of simulation models for computer subsystems, from CPUs and memory to network components. These models allow for simulations with different architectures and configurations. In this section, we utilize the REV-xBGAS CPU model and the *Merlin* network model to simulate xBGAS distributed systems across a range of network configurations [11]. This is done to showcase the simulator's capability for conducting functional correctness tests and facilitating performance comparisons.

A. Experiment setup

All simulations are executed on a production academic HPC cluster comprising 240 nodes. Each node is equipped with 64 physical AMD EPYC 7702 processors and boasts a memory capacity of 512 GB. These nodes operate on a CentOS 8 system running kernel version 4.18.0. The simulation tasks are submitted to the HPC cluster via Slurm version 22.05.8. The number of physical nodes engaged in the simulations varies depending on the scale of the xBGAS distributed system being simulated.

B. Benchmarks

We utilize two microbenchmarks implemented with the xBGAS runtime library for our analysis [3]. The microbenchmarks are compiled by the xBGAS RISC-V GNU compiler toolchain to translate the extended xBGAS instructions into binaries that can be recognized by the REV-xBGAS simulator. The first microbenchmark involves *broadcasting* four integers (16 bytes) from one processing element (PE) to others, while the second one entails each PE transmitting four integers to every other PE, known as the "*all-to-all*" communication. These microbenchmarks are instrumental in assessing the scalability of the xBGAS node count across diverse network topologies. It's noteworthy that both the broadcast and all-to-all operations encompass procedures to synchronize PEs, ensuring the completion of all remote memory operations before microbenchmarks terminate on all PEs.

C. Simulation parameters

All the simulated xBGAS nodes are standardized to share an identical node configuration. Each of these simulated nodes is configured with an REV-xBGAS CPU model that has the RISC-V RV64G architecture with the xBGAS extension. Each node is set to have a single 2.5 GHz processor and 1 GB of

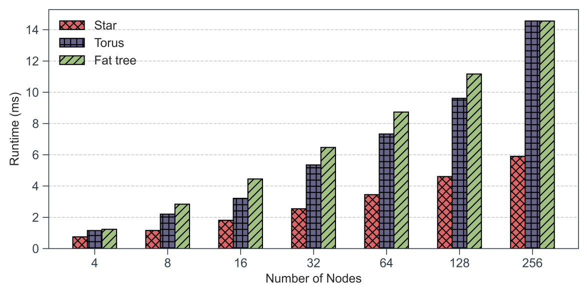

Figure 3. The Performance Comparision of the Broadcast Microbenchmark

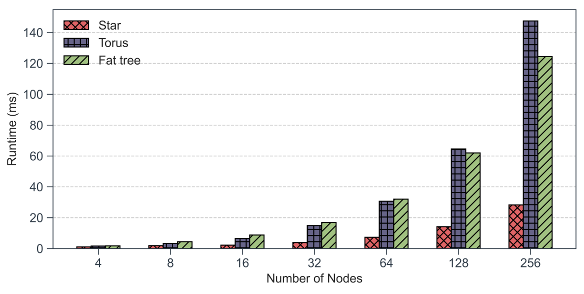

Figure 4. The Performance Comparision of the All-to-all Microbenchmark

memory, which is sufficient to execute microbenchmarks. Consequently, memory operations between Processing Elements (PEs) necessitate traversing the fabric and are managed by remote memory controllers.

Although REV is compatible with alternative memory hierarchy models such as MemHierarchy [12], known for its accurate memory simulation, we use the REV built-in memory model that prioritizes speed, albeit at the expense of memory modeling fidelity. This trade-off is sufficient for microbenchmarks characterized by small local memory footprints. The network module employs the Merlin library [11], which offers flexible networking components for simulating high-speed networks across different topologies. In our experimental setup, we configure the xBGAS distributed system with node counts ranging from 4 to 256. These configurations encompass various topologies, including *Star*, *Torus*, and *Fat Tree*. The specifics of the configuration are outlined in Table I.

D. Performance comparison

The performance results for the broadcast and all-to-all microbencharks are shown in Figure 3 and Figure 4, respectively. The y-axis shows the runtime in ms and the X-axis represents the number of nodes in the xBGAS distributed system configuration.

While the star topology isn't commonly employed in larger HPC systems, our experiments reveal its advantageous performance for smaller-scale setups. In both broadcast and all-to-all microbenchmarks, the star topology consistently demonstrates the shortest runtime when compared to torus and fat-tree topologies. In the case of broadcast, torus showcases marginally better runtime than fat-tree when node count remains below 128; however, they perform comparably as node count reaches 256. The all-to-all microbenchmark unveils a performance transition between torus and fat-tree: for node counts under 64, torus performs better; once the threshold is exceeded, fat-tree has the advantage.

V. Conclusion and Future Work

In this work, we present the design and implementation details of REV-xBGAS, the simulation infrastructure built atop the REV RISC-V CPU model, aiming to provide a cycle-based simulation environment for `xBGAS`. REV-xBGAS integrates customized and new components into REV, reflecting the architectural modifications necessary for simulating `xBGAS` at the cycle level. By leveraging the network models offered by SST, we configured the xBGAS-enabled distributed system with various network topologies. Subsequently, we conduct the execution of two microbenchmarks across different scales of xBGAS distributed systems. Our experiments not only demonstrate functional correctness but also facilitate performance comparisons across diverse system configurations.

As REV continues to undergo active development, our forthcoming efforts encompass integrating the latest upstream updates of REV into REV-xBGAS. We also intend to enhance the `xBGAS` runtime library by incorporating algorithms for collective operations tailored to specific network topologies. Meanwhile, our ongoing initiatives involve porting more benchmarks to `xBGAS`, within the REV-xBGAS simulation environment, with plans to share additional results with the community in the near future.

References

[1] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 1105–1114.

[2] J. D. Leidel, X. Wang, F. Conlon, Y. Chen, D. Donofrio, F. Fatollahi-Fard, and K. Keville, "xbgas: Toward a risc-v isa extension for global, scalable shared memory," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, 2018, pp. 22–26.

[3] B. Williams, X. Wang, J. D. Leidel, and Y. Chen, "Collective communication for the risc-v xbgas isa extension," in *Workshop Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[4] X. Wang, J. D. Leidel, B. Williams, A. Ehret, M. Mark, M. A. Kinsy, and Y. Chen, "xbgas: A global address space extension on risc-v for high performance computing," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 454–463.

[5] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.

[6] B. Keller, "Risc-v, spike, and the rocket core," *Berkeley Architecture Group*, 2013.

[7] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood, "The structural simulation toolkit: exploring novel architectures," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, pp. 157–es.

[8] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[9] T. C. Labs. (2020) xbgas architecture specification. [Online]. Available: https://github.com/tactcomplabs/xbgas-archspec/releases/tag/v0.0.6

[10] T. C. Labs. (2023) Rev: Risc-v native cpu model for sst. [Online]. Available: https://github.com/tactcomplabs/rev

[11] K. S. Hemmert, "Merlin element library deep dive." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[12] G. R. Voskuilen, "Sst deep dive: Memhierarchy." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.