SV-Sim: Scalable PGAS-Based State Vector Simulation of Quantum Circuits

Ang Li[†‡], Bo Fang[†‡], Christopher Granade[*], Guen Prawiroatmodjo[*], Bettina Heim[*], Martin Roetteler[*‡], Sriram Krishnamoorthy[†‡]
[†]{ang.li, bo.fang, sriram}@pnnl.gov, [*]{christopher.granade, guenp, bettina.heim, martinro}@microsoft.com
[†]Pacific Northwest National Laboratory, Richland, WA 99354, USA
[*]Microsoft Research, Redmond, WA 98052, USA
[‡]Quantum Science Center, Oak Ridge, TN 37931, USA

ABSTRACT

High-performance quantum circuit simulation in a classic HPC is still imperative in the NISQ era. Observing that the major obstacle of scalable state-vector quantum simulation arises from the massively fine-grained irregular data-exchange with remote nodes, in this paper we present SV-Sim to apply the PGAS-based communication models (i.e., direct peer access for intra-node CPUs/GPUs and SHMEM for inter-node CPU/GPU clusters) for efficient general-purpose quantum circuit simulation. Through an orchestrated design based on device functional pointer, SV-Sim is able to abstract various quantum gates across multiple heterogeneous backends, including IBM/Intel/AMD CPUs, NVIDIA/AMD GPUs, and Intel Xeon Phi, in a unified framework, but still asserting outstanding performance and tractable interface to higher-level quantum programming environments, such as IBM Qiskit, Microsoft Q# and Google Cirq. Circumventing the obstacle from the lack of polymorphism in GPUs and leveraging the device-initiated one-sided communication, SV-Sim can process circuit that are dynamically generated in Python using a single GPU/CPU kernel without the need of expensive JIT or runtime parsing, significantly simplifying the programming complexity and improving performance for QC simulation. This is especially appealing for the variational quantum algorithms given the circuits are synthesized online per iteration. Evaluations on the latest NVIDIA DGX-A100, V100-DGX-2, ALCF Theta, OLCF Spock, and OLCF Summit HPCs show that SV-Sim can deliver scalable performance on various state-of-the-art HPC platforms, offering a useful tool for quantum algorithm validation and verification. SV-Sim has been released at http://github.com/pnnl/sv-sim. A version specially tweaked for Q#/QDK is also provided.

KEYWORDS

Quantum Simulation, GPU, NVSHMEM, OpenSHMEM

ACM Reference Format:
Ang Li[†‡], Bo Fang[†‡], Christopher Granade[*], Guen Prawiroatmodjo[*], Bettina Heim[*], Martin Roetteler[*‡], Sriram Krishnamoorthy[†‡]. 2021. SV-Sim: Scalable PGAS-Based State Vector Simulation of Quantum Circuits. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476169

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476169

1 INTRODUCTION

Despite *Noisy-Intermediate-Scale-Quantum* (NISQ) [51] based quantum computing (QC) having achieved tremendous progress in the past few years [6, 10], to the broad QC community, the development and validation of quantum algorithms still need to mostly rely on simulation in a classical machine [27]. This is especially the case for the most promising quantum variational algorithms (VQA), such as variational quantum eigensolvers (VQE) [28, 52], quantum neural networks (QNN) [9, 19] (see the example in Figure 1), and quantum approximated optimization algorithms or QAOA [18, 20], because: (1) The scarce QC testbed resources such as IBM-Q are shared by a large group of users. The waiting time for resource allocation can be long. The iterative nature of variational algorithms further exacerbates the contention. Training a prototype 5-qubit QNN (e.g., Figure 1) can easily take days to finish. (2) These parameterized variational circuits are typically quite deep (e.g., 2.3M gates for a 24-qubit VQE-UCCSD circuit). The depth of the circuits can easily surpass the maximum gates allowed for the NISQ devices due to the short coherence time. (3) Present QC testbeds are not affordable for full-scale quantum-error-correction (QEC) [8, 45] thus incorporating high error rate [51]. To validate a quantum algorithm, or debug a circuit, simulation results are still necessary [12, 27].

For QC simulation, execution time is the primary performance metric, because: (i) The compute capacity or execution time required for simulating a quantum circuit is exponentially increased with the width of the circuit (i.e., number of qubits), and linearly increased with the depth of the circuit (i.e., number of gates); (ii) Emerging variational quantum algorithms iteratively execute the parameterized circuit to locate the ground state corresponding to the optimal solution. The deep circuit, the high error-rate, and the need to repeatedly sample from the resulting QC state (e.g., the probability of *c0* being 0 implies the binary classification result in Figure 1) further increase the number of executions required to verify a circuit. The speed of simulation thus directly associates with the cost of exploring these variational algorithms, ultimately affecting the efficiency of QC algorithm development.

Designing an efficient and scalable QC simulator on classical HPC systems, however, is quite challenging. The first obstacle is the memory bandwidth bound. According to the roofline model [58], the arithmetic intensity of QC simulation is less than 1/2 [22],

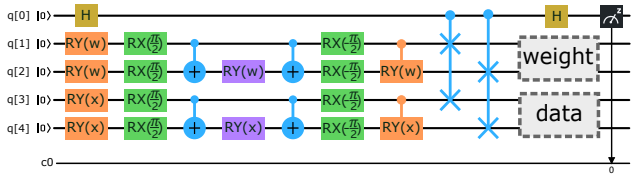

Figure 1: An exemplar variational QNN circuit for binary classification. The horizontal lines represent qubits. The blocks represent gates. The execution order is from left to right. Here, two qubits are used for data and two for weight. The rotation gates encode the classical information into the quantum circuit.

which means it is memory bound for most processors. However, due to large size, strided access, and streaming updates per gate, traditional wisdom in mitigating the memory bottleneck through cache hierarchy does not work well. The second obstacle is the communication hurdle. When the state vector cannot fit into the local memory, the strided memory access becomes massively fine-grained irregular communication among the processors and/or the nodes, leading to serious performance concerns for the traditional MPI-based coarse-grained communication methodology. Additionally, today's supercomputers are mostly heterogeneous [47]. Since the communication is handled by the CPUs, the data movement and synchronization between the CPUs and the accelerators can further exhaust the communication channels and drag the overall performance. The third obstacle is the computation efficiency of the accelerators. Regarding today's heterogeneous supercomputers, the performance of an application largely depends on how well the accelerators, such as GPUs, can be leveraged. Nevertheless, a unified and easy-to-use programming interface is still desired for domain users.

Existing QC simulators (see Section 6) do not handle the three aforementioned obstacles well, especially at large scale, partially because the three factors are inter-correlated, and the underlying reason is that *the accelerators are unable to perform instant and fine-grained inter-device/node communication on its own*. Therefore, the intensive irregular communication can easily offset the benefit from parallelization and scaling. For example, in order to harvest high computation and memory bandwidth, the processing of gates are offloaded to the accelerators such as GPUs. However, to exchange intermediate results among nodes, data has to be migrated from the accelerators to the system memory for transportation [60]. Alternatively, to improve inter-node communication efficiency, small messages such as single data fetches are typically packed for coarser-gained transportation, which leads to unnecessary waiting and wastes the opportunities to overlap communication and computation in a fine-grained manner [41]. Consequently, existing QC simulators cannot fully leverage the capability of the accelerators, especially at large scale. To the best of our knowledge, despite many works reporting GPU acceleration, most of them use a single GPU or GPUs in a node, barely any of them have been demonstrated in a large-scale GPU cluster [17, 44, 60]. This is also true for the widely used QC development environments such as IBM Qiskit [3], Google Tensorflow-Quantum/Cirq [11] and Microsoft QDK [46].

In this work, we investigate the novel PGAS-based communication models — *direct peer-access for intra-node CPU/GPU peers, SHMEM for CPU/GPU clusters* — for scalable state-vector QC simulation. We propose a new heterogeneous programming framework that can coordinate various backends in a unified form but still assert superior performance and easy-to-use programming interface. We demonstrate the SV-Sim simulator on various workstations and HPCs, including IBM/AMD/Intel CPUs, Intel Xeon Phi, and NVIDIA/AMD GPUs. Experimental results on OLCF Summit, ALCF Theta, NVIDIA V100 DGX-2 and DGX-A100 demonstrate the scalability and general usability of the simulator. Particularly, **using SV-Sim, the 16-GPU DGX-2 machine can simulate a 24-qubit 2.3M-gate VQE circuit in 3.5 mins**. We believe the proposed simulator can serve as a useful validation tool to further accelerate QC algorithm development nowadays.

This paper thus makes the following contributions:

- This is the first effort to investigate SHMEM communication models, including direct peer access and SHMEM for scalable state-vector quantum circuit simulation.
- This is the first state-vector quantum simulator reporting large-scale simulation in a GPU supercomputer like Summit.
- Evaluations on several state-of-art platforms demonstrate the performance, scalability and portability of SV-Sim.
- Being a promising communication technology in addition to MPI, our experience with PGAS-based OpenSHMEM, NVSHMEM and GPUDirect peer access can be beneficial for the development of alternative HPC applications.

2 BACKGROUND AND MOTIVATION

We describe the basic algorithm of state-vector quantum circuit simulation, explaining the communication issue. We also briefly introduce the PGAS-based SHMEM communication model, highlighting the motivation of using SHMEM for the simulation.

2.1 Quantum Circuit Simulation

Quantum circuit simulation is essentially to simulate the operation of applying a series of unitary operators $U_{m-1} \cdots U_1 U_0$ to a state vector $|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$ that describes the pure state of a quantum system where n is the number of qubits. Typically a complex-valued double-precision floating-point vector $\vec{\alpha}$ of size 2^n is used to store the coefficients α_i, which costs 16×2^n bytes of memory. U_i with $i \in [0, m-1]$ is a 2×2 or 4×4 complex-valued matrix. The size of 2×2 is for 1-qubit gate and 4×4 for 2-qubit gate, as it has been shown that an arbitrary quantum circuit can be decomposed into 1-qubit and 2-qubit gates [7]. Also note that the real quantum machine actually executes 1-qubit and 2-qubit gates internally [25].

To apply a gate U, the operation is $|\psi\rangle \rightarrow U |\psi\rangle$. For 1-qubit U applying on qubit q in a quantum register, $\vec{\alpha}$ is updated through:

$$\begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^q} \end{bmatrix} \rightarrow U_{2\times2} \cdot \begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^q} \end{bmatrix} \tag{1}$$

where $s_i = \lfloor i/2^q \rfloor 2^{q+1} + (i\%2^q)$ for every integer $i \in [0, 2^{n-1} - 1]$. Regarding 2-qubit U applying on qubit p and q (assuming $p < q$ without losing generality), $\vec{\alpha}$ is updated through:

$$\begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^p} \\ \alpha_{s_i+2^q} \\ \alpha_{s_i+2^p+2^q} \end{bmatrix} \rightarrow U_{4\times4} \cdot \begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^p} \\ \alpha_{s_i+2^q} \\ \alpha_{s_i+2^p+2^q} \end{bmatrix} \tag{2}$$

where $s_i = \lfloor \lfloor i/2^p \rfloor / 2^{q-p-1} \rfloor 2^{q+1} + (\lfloor i/2^p \rfloor \% 2^{q-p-1}) 2^{p+1} + (i \% 2^p)$ for every integer $i \in [0, 2^{n-2} - 1]$.

It can be seen that the stride of memory access from s_i to s_{i+1} really depends on the position of the target qubit (i.e., p or q), and is typically non-continuous. If $\vec{\alpha}$ can fit into the local memory, it implies many irregular memory access and inferior cache performance when considering 2^{n-1} or 2^{n-2} access per gate and m gates; if $\vec{\alpha}$ is shared among multiple devices of the same node or across different nodes, then it implies massively fine-grained inter-device communication. Such communication is so intensive that it can easily become the system bottleneck. Traditional approaches rely on CPU-managed MPI communication primitives for the data-exchange, introducing extra overhead for CPU/device data migration and frequent CPU/device execution switches (the cost per GPU kernel call can be as much as 20μs [29]). In order to increase the message granularity for better communication efficiency, extra packing & unpacking and unnecessary waiting are incurred as well. Both of them can lead to hardware under-utilization.

2.2 SHMEM Communication Model

The *Partitioned Global Address Space* (PGAS) based SHMEM communication model has been proposed since early 90s (first implemented for the Cray-T3D system), but was recently re-emerged as a promising alternative to the conventional MPI approach especially for GPUs, due to stronger support in emerging commodity interconnects [35, 36] such as NVLink and Infinity Fabric. As shown in Figure 2, it offers simple shared memory abstraction but still explicitly exposes the data locality information to the applications. It provides light-weight one-sided communication primitives that can minimize process synchronization and facilitate better overlapping between computation and communication especially for small messages. These features, together with the nice capability to initiate communication at the device side without aborting the kernel, makes it an ideal communication model for addressing the issues raised for the state-vector simulation in Section 2.1. We discuss the SHMEM model in terms of intra-node and inter-node conditions.

2.2.1 Intra-Node SHMEM model. For the intra-node condition, regarding CPUs, shared-memory is natural, as every core or socket of a CPU processor sees the same system memory. The caches may be local, but are transparent to the applications. Therefore, for intra-node CPUs, applications can directly access the unified memory using the same pointer. This is also true for Xeon-Phis [32, 54].

Regarding GPUs, NVIDIA GPUs allow direct peer-access through GPUDirect [53], provided the GPU pair is directly linked through NVLink or NVSwitch [35]. Based on our testing, accessing remote peer-GPU memory requires a remote pointer. Consequently, a PGAS-based SHMEM model can be manually constructed by sharing a pointer-array among all peer GPUs. We verified that for latest AMD GPUs such as MI100 that are directly connected by Infinity Fabric [5], a similar strategy works as well. Therefore, for intra-node GPUs, a PGAS communication model similar to SHMEM can be set up manually.

2.2.2 Inter-Node SHMEM Model. For the inter-node condition, regarding CPUs, OpenSHMEM [13] is the major effort that brings together different SHMEM implementations under an open standard for the parallel community. It operates on a symmetric memory address space where each *processing element* (PE) has its own local copy or partition of an array, but in the meanwhile can access other PE's copy using the same array name. In other words, the array is collectively allocated among all PEs. OpenSHMEM defines point-to-point (P2P) and collective communication (CL) primitives for data-exchange among PEs. Practicing the one-sided communication paradigm, the P2P `shmem_put` primitive, in the same semantics of memory store, immediately returns when the data has been copied out of the local memory, without waiting for the remote PE to finish the write. Similarly, the P2P `shmem_get` primitive, similar to memory load, returns only when the required data is available in the local memory, and does not need any synchronization with remote PE. Atomic access and locks are also provided for critical regions. To summarize, OpenSHMEM is highlighted for its one-sided, RDMA-supported, single-process-multiple-data (SPMD) interfaces for *low-latency*, *fine-grained* data-exchange in distributed-memory HPC systems. An implementation of OpenSHMEM is distributed with OpenMPI [55], which we use in our evaluation.

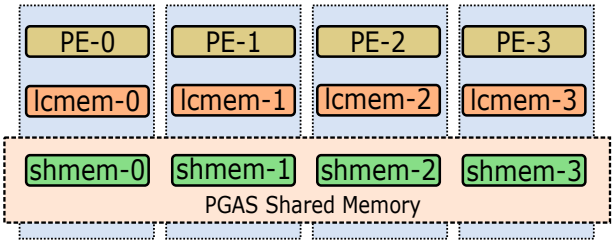

Figure 2: PGAS-based shared memory communication model. "lcmem" refers to the local memory per-PE. "shmem" refers to the shared memory among PEs.

Regarding NVIDIA GPUs, the latest NVIDIA HPC SDK releases NVSHMEM [48, 50], which is a GPU implementation of the OpenSHMEM standard with some GPU-specific extension. It provides CUDA kernel-side primitives that enable GPU threads to access arbitrary locations in a symmetrically-distributed memory of the same node or across different nodes. It offers fine-grained, low-overhead remote data access from inside the kernel, and benefits performance with the latency-hiding capability of the warp hardware scheduler [48]. Importantly, NVSHMEM supports MPI/NVSHMEM mixed programming well as it allows using an MPI global communicator to initialize the NVSHMEM process. However, the MPI/OpenSHMEM mixed programming requires the initialization of both MPI and OpenSHMEM, and then reform MPI communicator with OpenSHMEM node indices. In our testing, this occasionally encounters MPI non-initialization issues. NVSHMEM also gains performance from GPUDirect-RDMA [53] and the NCCL library [26].

Regarding AMD GPUs, a prototype implementation known as *ROC_SHMEM* is available [4]. The design detail is presented in [21]. Basically it aligns with the common initiative from the major GPU vendors (NVIDIA and AMD) about GPU-centric networking accessible from within the GPU kernel. ROC_SHMEM offers an OpenSHMEM-like interface that simplifies the programming complexity while encouraging fine-grained computation/communication overlap over host-driven networking.

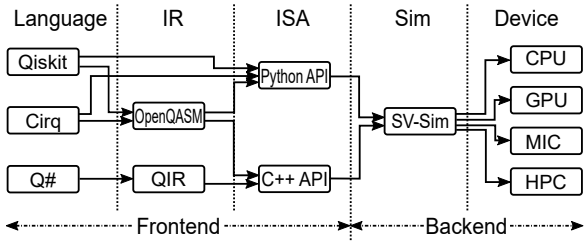

Figure 3: SV-Sim Framework.

3 UNIFIED SV-SIM FRAMEWORK

We present the design goals and overall SV-Sim framework. We then discuss SV-Sim backend and frontend in detail.

3.1 SV-Sim Design Goals and Structure

Frontend Requirements: Supporting mainstream QC programming environments such as Qiskit, Cirq and Q#.

Backend Requirements: Supporting various HPC hardware platforms, including CPUs, GPUs, MICs, in a unified framework. Hiding the implementation details (e.g., CUDA, HIP, OpenMP, OpenSHMEM, NVSHMEM) but still allowing device-specific optimization.

Performance Requirements: Demonstrating scalable performance advantage on various backends by fully leveraging accelerator capability and the PGAS-based communication model.

Flexibility Requirements: To offer flexibility for verifying quantum circuits that are dynamically synthesized at runtime, such as the variational algorithms, while conserving high-performance, the entire circuit should be simulated altogether in a single kernel. However, we try to avoid recompilation at runtime.

Figure 3 illustrates the SV-Sim framework. SV-Sim provides both C/C++ and Python interface to support quantum programming environments including Qiskit, Criq and Q#, as well as intermediate representation such as OpenQASM and QIR. The backends include CPU, GPU and Xeon-Phi in terms of single-device, single-node scale-up and multi-nodes scale-out.

3.2 SV-Sim Backend Design

We discuss single-device, single-node scale-up and multi-nodes scale-out, respectively.

3.2.1 SV-Sim Single-Device Design. The single-device SV-Sim mainly achieves the performance & flexibility goals through: (a) homogeneous execution; (b) specialized gate implementation.

Homogeneous execution: Homogeneous execution means all the application logic such as the simulation of the target quantum circuit is executed as a single function for the device, like CPU or GPU, in order to avoid unnecessary data migration and context switch [29]. Particularly for GPUs, we try to compact all the gate functions into a single GPU kernel. However, this compaction cannot be done at runtime to avoid expensive JIT considering performance. Additionally, both CUDA and HIP do not support polymorphism, while parsing & dispatching the gate on the GPU-side at runtime are very costly (as will be seen later). In this work, **we find a novel way in CUDA to achieve polymorphism effectively through device functional pointers**, as shown in Listing 1. Essentially, each CUDA device function has its unique function address in the

```
1 //define functional pointer type
2 using func_t=void (*)(const Gate*,ValType*,ValType*);
3 //declare GPU device funtional pointer
4 extern __device__ func_t pT_OP;
5 class Gate{
6   ...
7   Gate* upload() {
8     //assign device function in constant memory to the pointer
9     cudaSafeCall(cudaMemcpyFromSymbol(&op,pT_OP,sizeof(func_t)));
10  }
11  __device__ void exe_op(ValType* sv_real,ValType* sv_imag){
12    (*(this->op))(this,sv_real,sv_imag);//execute gate function
13  }
14  func_t op;//gate function pointer
15 }
16 __device__ void T_OP(Gate* g,ValType* sv_real,ValType* sv_imag){
17     T_GATE(sv_real, sv_imag, g->qb0);
18 }
19 //define GPU device functional pointer
20 __device__ func_t pT_OP = T_OP;
21 __global__ void simulation_kernel(Simulation* sim){
22     grid_group grid = this_grid(); //for global synchronization
23     //execute all gates in the circuit
24     for (IdxType t=0; t<(sim->n_gates); t++)
25         ((sim->circuit_gpu)[t])->exe_op(sim->sv_real,sim->sv_imag);
26 }
```

Listing 1: Functional pointer based simulation framework.

constant memory for a particular GPU. When a circuit is conveyed from a frontend interface, depending on gate type (e.g., a T-gate), we can copy the address of the corresponding gate device function to a uniform functional pointer inside the gate object (Line 9). Therefore, the gate function can be directly invoked during execution (Line 12). In this way, on one hand we avoid runtime parsing & branching, while on the other hand we avoid generating a giant kernel with massive functions being inlined which can incur long compile time [41]. The functional pointer achieves polymorphism by unifying the format of the gate function and its parameters (Line 12). Nevertheless, in practice we observe that the latency for `cudaMemcpyFromSymbol` can be non-trivial. Consequently, we preload these gate device functional pointers as member functional pointers of the simulation object during environment initialization, and then directly copy a member functional pointer to a gate (thus purely CPU operations). In this way, we significantly reduce the number of calls to `cudaMemcpyFromSymbol` (i.e., from the number of gates of a circuit to a fixed number of gates being supported). We verify that this functional pointer approach also works for CPUs. Therefore, through this, we can build a unified simulation framework for both CPUs and GPUs.

For the current HIP on AMD GPUs, however, this GPU device functional pointer feature is not supported yet. Therefore, we have to rely on gate parsing and dispatching at runtime. This brings some performance concerns. Additionally, the current HIP runtime does not support recursive device function inlining. We find that in the hierarchical calling graph, the leaf device functions must be declared as *__noinline__*, while all the intermediate branch functions should be inlined. Being unable to inline all the device function calls may also bring some performance slow-down. Queries with AMD ROCM developers confirmed future support for both features.

Specialized gate implementation: Different from existing simulators like *Aer* in Qiskit and *qsim* in Tensorflow-Quantum adopting a generalized 1-qubit or 2-qubit unitary gate implementation for all the gates, we observe that: (1) *A particular gate may allow gate-specific optimization.* For example, a 1-qubit T-gate is in the form $T = \frac{1}{\sqrt{2}}[[1,0][0,1+i]]$. Therefore, we only need the calculation

```
1 void T_GATE(ValType* sv_real,ValType* sv_imag,const IdxType qubit){
2   const __m512d s2i_v=_mm512_set1_pd(S2I);
3   __m256i idx=_mm256_set_epi32(0,1,2,3,4,5,6,7);
4   const __m256i inc=_mm256_set1_epi32(8);
5     for(IdxType i=0;i<half_dim;i+=8,idx=_mm256_add_epi32(idx,inc)){
6       __m256i pos1 = ...;
7       const __m512d el1_real=_mm512_i32gather_pd(pos1,sv_real,8);
8       const __m512d el1_imag=_mm512_i32gather_pd(pos1,sv_imag,8);
9       // sv_real[pos1]=1/sqrt(2)*(el1_real-el1_imag);
10      __m512d tmp0=_mm512_sub_pd(el1_real,el1_imag);
11      __m512d tmp1=_mm512_mul_pd(s2i_v,tmp0);
12      _mm512_i32scatter_pd(sv_real,pos1,tmp1,8);
13      // sv_imag[pos1]=1/sqrt(2)*(el1_real+el1_imag);
14      tmp0=_mm512_add_pd(el1_real,el1_imag);
15      tmp1=_mm512_mul_pd(s2i_v,tmp0);
16      _mm512_i32scatter_pd(sv_imag,pos1,tmp1,8);
17    }
18 }
```

Listing 2: SV-Sim single-device T-gate design with AVX512.

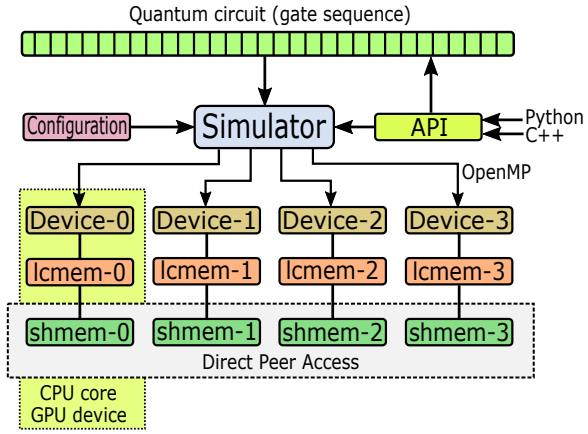

Figure 4: SV-Sim single-node scale-up architecture.

for the last element $1 + i$, saving more than half of the computation and memory access. We apply similar gate-specific optimization for other gate functions. (2) *A particular backend may allow architecture-specific optimization.* Listing 2 shows an example of the T-gate optimized with AVX512 vectorization on Xeon-Phi KNL [54] and recent Intel CPUs with AVX512. As can be seen, each step we proceed with 8 elements in the state-vector (Line 5), as each 512 bits can accommodate 8 double-precision operands. Since the loading data and storing data for the 8 vector lanes are not necessarily continuous in memory, we use `gather` and `scatter` primitives for data loading (Line 7-8) and storing (Line 16). We also apply GPU-specialized implementation for the GPU designs.

3.2.2 SV-Sim Single-Node Scale-up Design. **Architecture:** Figure 4 shows the overall architecture for the scale-up implementation. Similar to the single-device condition, the gate objects from the frontend interface are buffered in a circuit queue. Before the simulation starts, the entire circuit buffer is transferred to the device-side. For CPUs, the circuit queue is shared among all the CPU cores since it is read-only. As all the CPU cores share the same system memory space, they can access arbitrary memory locations using the same memory pointer. For GPUs, the circuit has to be duplicated in each device since each gate object contains a per-GPU unique device functional pointer. For each GPU, the memory space is partitioned; they use their local memory (i.e., *lcmem* in Figure 5) for normal computation. The state-vector is evenly partitioned among all GPUs in a shared memory space, as discussed in the following.

```
1 void T_GATE(ValType* sv_real,ValType* sv_imag,const IdxType qubit){
2   _Pragma("omp for schedule(auto)") //scheduling policy
3   for (IdxType i=0; i<half_dim;i+=1){
4     pos1 = ...
5     const ValType el1_real = sv_real[pos1];
6     const ValType el1_imag = sv_imag[pos1];
7     sv_real[pos1] = S2I*(el1_real-el1_imag);
8     sv_imag[pos1] = S2I*(el1_real+el1_imag);
9   }
10  _Pragma("omp barrier") //for synchronization
11 }
```

Listing 3: SV-Sim scale-up T-gate design for CPU.

```
1 __device__ __noinline__ void T_GATE(ValType** sv_real_ptr,
2   ValType** sv_imag_ptr,const IdxType qubit){
3   multi_grid_group grid = this_multi_grid();
4   for (IdxType i=grid.thread_rank(); i<half_dim;
5         i+=grid.size()){
6     IdxType pos = ...
7     IdxType pos1_grid = pos / sv_num_per_gpu;
8     IdxType pos1 = pos % sv_num_per_gpu;
9     const ValType el1_real = sv_real_ptr[pos1_gid][pos1];
10    const ValType el1_imag = sv_imag_ptr[pos1_gid][pos1];
11    sv_real_ptr[pos1_gid][pos1] = S2I*(el1_real-el1_imag);
12    sv_imag_ptr[pos1_gid][pos1] = S2I*(el1_real+el1_imag);
13  }
14  grid.sync();
15 }
16 ...
17 ValType** sv_real_ptr_cpu; ValType** sv_imag_ptr_cpu;
18 ValType** sv_real_ptr; ValType** sv_imag_ptr;
19 SAFE_ALOC_HOST(sv_real_ptr_cpu, sizeof(ValType*)*n_gpus);
20 SAFE_ALOC_HOST(sv_imag_ptr_cpu, sizeof(ValType*)*n_gpus);
21 SAFE_ALOC_GPU(sv_real_ptr, sizeof(ValType*)*n_gpus);
22 SAFE_ALOC_GPU(sv_imag_ptr, sizeof(ValType*)*n_gpus);
23 for (unsigned d=0; d<n_gpus; d++){
24   hipSafeCall(hipSetDevice(d));
25   SAFE_ALOC_GPU(sv_real_ptr[d],sv_num_per_gpu*sizeof(ValueType));
26   SAFE_ALOC_GPU(sv_imag_ptr[d],sv_num_per_gpu*sizeof(ValueType));
27   for (unsigned g=0; g<n_gpus; g++)
28     if (g != d) hipSafeCall(hipDeviceEnablePeerAccess(g,0));
29 }
30 ...
31 hipSafeCall(hipMemcpy(sv_real_ptr,sv_real_ptr_cpu,
32   sizeof(ValueType*)*n_gpus, hipMemcpyHostToDevice));
33 hipSafeCall(hipMemcpy(sv_imag_ptr,sv_imag_ptr_cpu,
34   sizeof(ValueType*)*n_gpus, hipMemcpyHostToDevice));
35 ...
36 #pragma omp parallel num_threads (n_gpus) shared(params)
37 {
38   int d = omp_get_thread_num();
39   hipSafeCall(hipSetDevice(d));
40   ...
41   if(d==0) hipLaunchCooperativeKernelMultiDevice(params,n_gpus,0);
42 }
```

Listing 4: SV-Sim scale-up design for AMD GPU using HIP.

Runtime: We use parallel OpenMP threads to manage the CPU cores and GPU devices at runtime. Therefore, the key is workload partition strategy and/or scheduling policy. For CPUs, we parallelize at the "*for*" loop (Line 2 in Listing 3), which comprises 2^{n-1} iterations regarding the 1-qubit condition. We use the "*auto*" scheduling policy, which reports the best performance across different policies. Based on our testing, a suboptimal policy like *dynamic* can drag the performance by more than two orders. A synchronization barrier is needed at the end to ensure data consistency across the loops of consecutive gates due to parallel execution.

For GPUs, we perform a more coarse-grained partition so as to facilitate coalesced memory access and exploit spatial cache locality [38]. The shared memory space is achieved through GPUDirect peer access, which allows a GPU to access peer GPU's memory inside a kernel through a remote memory pointer, provided the two GPUs are directly connected by NVLink, NVSwitch, or Infinity Fabric. Therefore, a novel PGAS communication model can be constructed

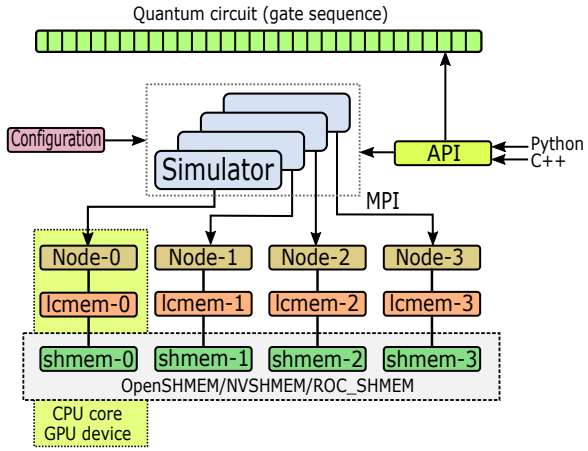

Figure 5: SV-Sim multi-nodes scale-out architecture.

by partitioning the state-vector evenly among all GPUs following the natural array order. Each GPU holds one such portion with a unique pointer. These pointers are collected in a pointer array, which is then broadcasted to all GPUs for remote referencing. Both NVIDIA and AMD GPUs support GPUDirect peer access.

Listing 4 shows the implementation logic for the T-gate in HIP for AMD GPUs. Again, since it is the leaf device function, it needs to be no-inlined in order to be compiled. Line 3 is to initialize a multi-device grid for multi-device synchronization in Line 14. This requires a specialized GPU kernel invocation approach in Line 41 using a single CPU thread. Based on the data position, we calculate which GPUs hold the data (Line 7) and what is the relative index (Line 8). Then, the remote memory can be fetched and stored in Line 8-12. We show the way to set up the pointer array in Line 17-34. Despite two GPUs being directly connected, we still need to enable GPUDirect peer access in Line 28 for each GPU pair. We use one OpenMP thread per GPU to manage the parallel execution (Line 36).

3.2.3 SV-Sim Multi-Nodes Scale-out Design. **Architecture:** Figure 5 illustrates architecture for the SV-Sim scale-out implementation based on SHMEM. The difference here with the scale-up condition is that each SHMEM (or MPI) process owns a simulator object. Therefore, rather than using a single simulator instance to manage multi-devices, each of them operates on a single device.

Runtime: We use SHMEM processes to manage the devices across multiple nodes, each process per CPU core or GPU device. Based on the size, we partition the state-vector arrays evenly among all SHMEM processing elements. We allocate the state-vector in the SHMEM shared space, and leverage the fine-grained "*put*" and "*get*" primitives for remote data storing and loading.

Listing 5 shows the implementation logic for T-gate using NVSHMEM for GPUs from multi-nodes. Different from the HIP implementation, the CUDA device function here is inlined (Line 1). We still need the cooperative grid object (Line 3) for thread grid synchronization in a GPU (Line 14). The reason is that the NVSHMEM global synchronization primitive `nvshmem_barrier_all()` can only be called by one thread per NVSHMEM node (Line 13). The synchronization across the NVSHMEM processes are also only for these representative threads, rather than the entire CUDA threads

```
1 __device__ __inline__ void T_GATE(ValType* sv_real,
2   ValType* sv_imag, const IdxType qubit){
3   grid_group grid = this_grid();
4   for (IdxType i=(sim->i_gpu)*per_pe_work+tid;
5     i<(sim->i_gpu+1)*per_pe_work; i+=blockDim.x*gridDim.x){
6     ValType el1_real=nvshmem_double_g(&sv_real[pos1],pos1_gid);
7     ValType el1_imag nvshmem_double_g(&sv_imag[pos1],pos1_gid);
8     ValType sv_real_pos1=S2I*(el1_real-el1_imag);
9     ValType sv_imag_pos1=S2I*(el1_real+el1_imag);
10    nvshmem_double_p(&sv_real[pos1], sv_real_pos1, pos1_gid);
11    nvshmem_double_p(&sv_imag[pos1], sv_imag_pos1, pos1_gid);
12  }
13  if(threadIdx.x==0&&blockIdx.x==0) nvshmem_barrier_all();
14  grid.sync();
15 }
16 ...
17 nvshmemx_init_attr_t attr;
18 MPI_Comm comm = MPI_COMM_WORLD;
19 attr.mpi_comm = &comm;
20 nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, &attr);
21 cudaSafeCall(cudaSetDevice(0));
22 ...
23 sv_real = (ValType*)nvshmem_malloc(sv_size_per_gpu);
24 sv_imag = (ValType*)nvshmem_malloc(sv_size_per_gpu);
25 ...
26 nvshmemx_collective_launch((const void*)simulation_kernel,gridDim,
27                THREADS_PER_BLOCK,args,0,0));
```

Listing 5: SV-Sim scale-out design for NVIDIA GPUs using CUDA and NVSHMEM.

Table 1: Gates defined in IBM OpenQASM standard.

Gates	Meaning	Gates	Meaning
U3	3 parameter 2 pulse 1-qubit	CY	Controlled Y
U2	2 parameter 1 pulse 1-qubit	SWAP	Swap
U1	1 parameter 0 pulse 1-qubit	CH	Controlled H
CX	Controlled-NOT	CCX	Toffoli
ID	Idle gate or identity	CSWAP	Fredkin
X	Pauli-X bit flip	CRX	Controlled RX rotation
Y	Pauli-Y bit and phase flip	CRY	Controlled RY rotation
Z	Pauli-Z phase flip	CRZ	Controlled RZ rotation
H	Hadamard	CU1	Controlled phase rotation
S	sqrt(Z) phase	CU3	Controlled U3
SDG	Conjugate of sqrt(Z)	RXX	2-qubit XX rotation
T	sqrt(S) phase	RZZ	2-qubit ZZ rotation
TDG	Conjugate of sqrt(S)	RCCX	Relative-phase CXX
RX	X-axis rotation	RC3X	Relative-phase 3-controlled X
RY	Y-axis rotation	C3X	3-controlled X
RZ	Z-axis rotation	C3XSQRTX	3-controlled sqrt(X)
CZ	Controlled phase	C4X	4-controlled X

of the kernel. We use "`double_g`" & "`double_p`" for data loading and storing in double-precision (Line 6-7 and Line 10-11). Accordingly to [48] and NVSHMEM developers, enhanced communication efficiency can be achieved if the remote access are coalesced per warp. Line 17-21 shows how to initialize NVSHMEM using MPI global communicator. Note, device id needs to be set to 0 since each process only sees one GPU. Line 23-24 show how to allocate memory in the shared space of NVSHMEM. Line 26-27 show how to launch a NVSHMEM kernel. For OpenSHMEM towards CPUs, and ROC_SHMEM towards AMD GPUs, the approaches are similar.

3.3 SV-Sim Frontend Design

In Figure 3 we show the stack of SV-Sim frontend components. We discuss it in detail here.

3.3.1 SV-Sim ISA. SV-Sim ISA is basically the gate-set internally implemented in SV-Sim. Table 1 lists the gates defined in the IBM OpenQASM specification [14], which includes 5 *basic gates* (column-wise from `U3` to `ID` in Table 1) that are natively executed by the IBM-Q quantum machines, 11 *standard gates* (from `X` to `RZ`) that are defined atomically (which are translated into basic gates during

Table 2: Gates defined in Microsoft QIR-Runtime standard.

Gates	Meaning	Gates	Meaning
X	Same as X in Table 1	ControlledZ	Same as CZ if 1-control
Y	Same as Y in Table 1	ControlledH	Same as CH if 1-control
Z	Same as Z in Table 1	ControlledS	Controlled S
H	Same as H in Table 1	ControlledT	Controlled T
S	Same as S in Table 1	ControlledR	Controlled R
T	Same as T in Table 1	ControlledExp	Controlled Exp
R	Unified rotation gate	AdjointT	Same as TDG in Table 1
Exp	Exponential of Pauli	AdjointS	Same as SDG in Table 1
ControlledX	Same as CX if 1-control	ControlledAdjointS	Controlled SDG
ControlledY	Same as CY if 1-control	ControlledAdjointT	Controlled TDG

IBM-Q machine-specific assembling phase), and 18 *compound gates* that are constituted by basic and standard gates. The SV-Sim backend internally implements the OpenQASM basic gates and standard gates through specialized gate implementation (see Section 3.2.1). The compound gates are realized by composing the call to basic gates and standard gates. In this way, SV-Sim supports OpenQASM quantum IR. Given several quantum programming environments can generate OpenQASM, such as Qiskit, ProjectQ, Scaffcc, Cirq, SV-Sim can support these environments through OpenQASM.

Additionally, to support Q#/QDK [56], SV-Sim also implements the gate set defined in the QIR-runtime. QIR [24] is a newly released LLVM-compatible IR proposed by Microsoft, aiming at providing a standard IR as a common intermediate abstraction for different frontend languages and backend machines. The simulator template included in the QIR runtime defines a set of gate function APIs (see Table 2) that if a user-defined simulator can concertize and realize these virtual functions, it can support Q# through QIR and QIR-runtime. The QIR-runtime gate set includes elementary gates (column-wise from X to Exp in Table 2), (multi-)controlled gates (from ControlledX to ControlledExp), and adjoint gates. We developed a wrapper in C++ to connect SV-Sim to QIR-runtime, and linked them altogether (including the QIR application code) in the compilation procedure. In this way, Q# is supported by SV-Sim through QIR, the wrapper and the C++ interface.

To directly support high-level programming environments such as Qiskit and Cirq rather than through an IR like OpenQASM, we develop a Python interface for SV-Sim using the *pybind11* package. To allow the multi-node SV-Sim based on SHMEM working with the HPC workload management environment through the Python API, we rely on *mpi4py* and NVSHMEM/MPI mixed programming to convey the MPI runtime information from Python to SV-Sim. This usually requires a specially compiled Python library, e.g., in ORNL Summit. For all the SHMEM models (OpenSHMEM, NVSHMEM and ROC_SHMEM, etc.), a library such as *shmem4py* is highly desired for Python support, especially in a distributed environment such as an HPC cluster.

4 EVALUATION

We evaluate SV-Sim on a variety of platforms, covering different architectures (i.e., CPU, GPU, and Xeon-Phi) and vendors (i.e., Intel/IBM/AMD CPUs, and NVIDIA/AMD GPUs). The platforms are summarized in Table 3. We evaluate SV-Sim in three categories: *single-device*, *single-node multi-devices*, and *multi-nodes*. For GPUs, we use CUDA and HIP events for the time measurement. For CPUs, we rely on the system time measurement approach. The reported results are the average values of 10 times' execution.

Regarding the quantum algorithms, we use applications from an OpenQASM benchmark suite QASMBench [31]. We use 8 medium-sized quantum circuits for single-device, and single-node-multi-device evaluation (see the upper part of Table 4). We use 8 large-sized circuits for multi-node evaluation (lower part of Table 4).

4.1 Single Device

In the single device evaluation, we test SV-Sim performance using the 8 medium-scale circuits on a high-end Intel CPU — Xeon Platinum-8276M with and without AVX-512 optimization, a latest AMD CPU — 2nd Gen EPYC-7742, an IBM Power-9 CPU, an Intel Xeon Phi-7230 accelerator with and without AVX512 optimization, an NVIDIA Volta V100 GPU, an NVIDIA Ampere A100 GPU, and an AMD MI100 GPU. The relative execution latency of each platform, respecting the latency of AMD EPYC-7742, are illustrated in Figure 6. The CPU results here are obtained for sequential execution using one CPU core. The observations are that: (i) With fewer qubits (n=11 or 12), CPUs show better performance than GPUs, due to good cache performance and the under-utilization of GPU cores [30]. However, with more qubits (n=13-15), GPUs show much better performance than CPUs, especially for NVIDIA V100 and A100 with more than 10X speedups; (ii) Applying AVX-512 optimization can generally bring ~2X on both Intel CPU and the Phi accelerator; (iii) We do not observe significant speedup from V100 to A100 for SV-Sim, which is likely because the major performance bottleneck is memory access bandwidth rather than computation; (iv) The single-core performance of Xeon-Phi is worse than that of a CPU as expected, because the many-cores in Xeon-Phi is light-weight Atom cores; (v) MI-100 shows suboptimal performance is likely because that, since the current rocm runtime does not support functional pointers, we have to perform the costly gate-parsing and branching process within the GPU-kernel per-gate. Unable to inline all the gate implementation also leads to a fat kernel showing reduced performance for the instruction cache.

4.2 Scale-Up: Single-Node-Multiple-Devices

CPU: We first evaluate the OpenMP-based SV-Sim on the Intel Xeon Platinum-8276M multi-core CPU with AVX512 optimization and using the unified memory space for inter-core communication. We scale the number of CPU cores used in the simulation from 1 to 256, and show the relative latency in Figure 7. As can be seen, (i) with qubits less than 15, increasing the number of cores does not accelerate the simulation due to parallelization & communication overhead and hardware under-utilization with insufficient workload. However, with 15 qubits, parallelization can gain performance with more than 2X; (ii) The optimal performance is obtained typically with 16 or 32 cores; (iii) Using more than 128 cores essentially imposes significant overhead, mostly from the communication contention in the QPI bus between the CPU sockets.

Xeon Phi: The manycore accelerator Xeon-Phi Knights Landing (KNL) follows a similar pattern as the CPU. The strong scaling result tested using a single Xeon Phi-7230 node in the ANL Theta HPC is shown in Figure 8. We adjust the number of cores from 1 until 64. Analogously, the performance sweet-point is in the middle of the spectrum – two cores for small problems (n=11 or 12), and four cores for large ones (n=13-15). This implies that, compared to

Table 3: Evaluation platforms. "Acc" refers to accelerators.

Platform	Nodes	CPU	Cores/Node	Mem	Compiler	Acc	Acc No.	Acc Mem	Interconnect	Acc Compiler
Intel Server	1	Intel Xeon P-8276M	224	6.34TB	gcc-9.2.0	N/A	N/A	N/A	N/A	N/A
A100 Server	1	AMD EPYC 7742	64	1.057TB	gcc-9.1.0	Ampere-A100 GPU	8	40GB HBM	NVLink & NVSwitch	cuda-11.1
V100-DGX-2	1	Intel Xeon P-8168	48	1.583TB	gcc-9.1.0	Volta-V100 GPU	16	32GB HBM	NVLink & NVSwitch	cuda-11.1
OLCF Spock HPC	36	AMD EPYC 7662	256	256GB	gcc-9.2.0	MI100 GPU	4	32GB HBM	Infinity Fabric	rocm-4.10
OLCF Summit HPC	4608	IBM Power-9	44	6.342GB	gcc-9.1.0	Volta-V100 GPU	6	16GB HBM	NVLink	cuda-11.2
ALCF Theta HPC	4392	Intel Xeon Phi-7230	64	198GB	gcc-9.3.0	Xeon Phi-7230	1	198GB	Omni-Path	gcc-9.3.0

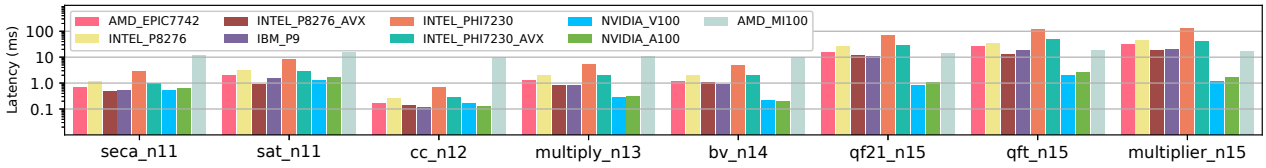

Figure 6: SV-Sim single-device execution latency on evaluated platforms (absolute latency in ms).

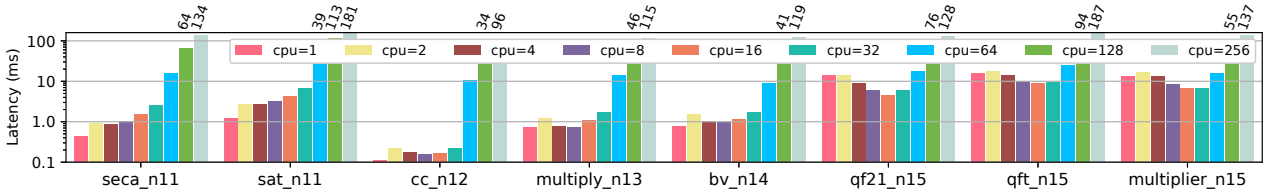

Figure 7: SV-Sim scaling-up evaluation on an Intel P8276M CPU via unified space with AVX512.

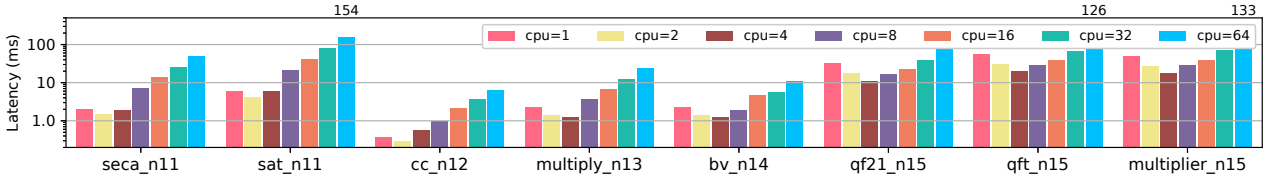

Figure 8: SV-Sim scaling-up evaluation on an ALCF Xeon-Phi7230 Theta node via unified space with AVX512.

Table 4: Quantum routines evaluated for SV-Sim.

Routine	Description	Qubits	Gates	CX	Category
seca	Shor's error correction code for teleportation	11	216	84	medium
sat	Boolean satisfiability problem	11	679	252	medium
cc	Counterfeit-coin finding algorithm	12	22	11	medium
multiply	Performing 3×5 in a quantum circuit	13	98	40	medium
bv	Bernstein-Vazirani algorithm	14	41	13	medium
qf21	Quantum phase estimation to factor 21	15	311	115	medium
qft	Quantum Fourier transform	15	540	210	medium
multiplier	Quantum multiplier	15	574	246	medium
dnn	quantum neural network sample	16	2016	384	large
bigadder	Quantum ripple-carry adder	18	284	130	large
cc	Counterfeit-coin finding algorithm	18	34	17	large
square_root	Get the square root via amplitude amplification	18	2300	898	large
bv	Bernstein-Vazirani algorithm	19	56	18	large
qft	Quantum Fourier transform	20	970	380	large
cat_state	Coherent superposition with opposite phase	22	22	21	large
ghz_state	Greenberger-Horne-Zeilinger state	23	23	22	large

the QPI bus between CPU sockets, the communication overhead triggered in the Omni-path 2D-mesh interconnect of Xeon Phi KNL [54] is even more prominent, which may suggest more constraint bandwidth for the all-to-all communication in KNL's 2D-mesh NoC than in QPI.

GPU: The condition for GPU is significantly different. Figure 9 shows the relative latency with the number of GPUs increased from 1 to 2, 4, 8 and 16 in the V100-DGX-2 system. The GPU peers are connected by NVSwitch, allowing all-to-all communication and GPUDirect peer access. Overall, the execution latency follows strong scaling for all the tested cases, except that for the small problems (n=11 and 12), there is slightly slow-down due to the introduction of communication when scaling from single-GPU to dual-GPUs. The 16-GPU execution achieves on-average more than 10.6X over the single-GPU version. Figure 10 shows the scaling-up results tested on DGX-A100. As can be seen, the trend is similar to DGX-2. However, from 4 GPUs to 8 GPUs, there is significant performance improvement.

Unlike NVIDIA V100-DGX-2 or DGX-A100, in the AMD workstation with 4 MI-100 GPUs connected by infinity fabric, the scaling appears to be linear and modest, as shown in Figure 11. Particularly, we do not observe the parallelization lag from single to dual-GPUs, as observed in V100-DGX-1. This is probably because for AMD GPUs, the bottleneck is in the computation kernel (as explained in Section 4.1) rather than in the communication fabric.

4.3 Scale-Out: Multiple-Nodes HPC Cluster

CPU OpenSHMEM: For the HPC cluster execution, we test the 8 quantum circuits from the large category (Table 4) on the IBM Power-9 CPUs in the OLCF Summit supercomputer. As each node has 44 hardware threads, we specify 32 cores per resource-set, and scale the OpenSHMEM nodes from 32 until 1024. In other words, performance drag can be observed from 32 cores of intra-node to 64 cores across nodes. This is observed for `cc_n18` and `bv_n19` in Figure 12. Other than that, the performance scaling is mostly linear or incremental. From 32 to 1024 cores, the latency reduction is less than 3X. This is mainly because the bottleneck is not the computation but the communication, since the all-to-all

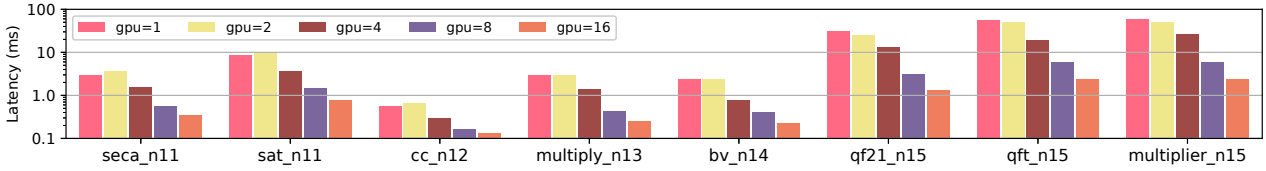

Figure 9: SV-Sim scaling-up evaluation on an NVIDIA V100 DGX-2 workstation using direct GPU-Direct peer-access.

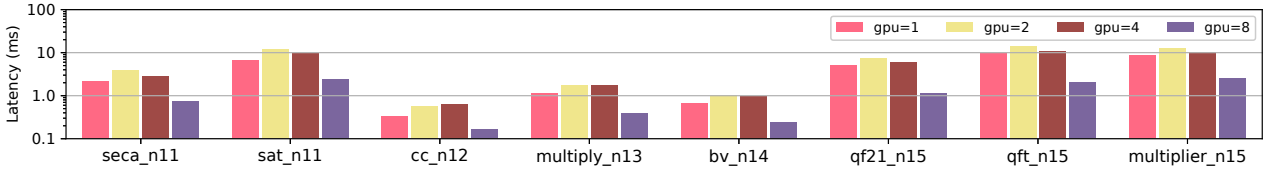

Figure 10: SV-Sim scaling-up evaluation on an NVIDIA DGX-A100 workstation using direct GPU-Direct peer-access.

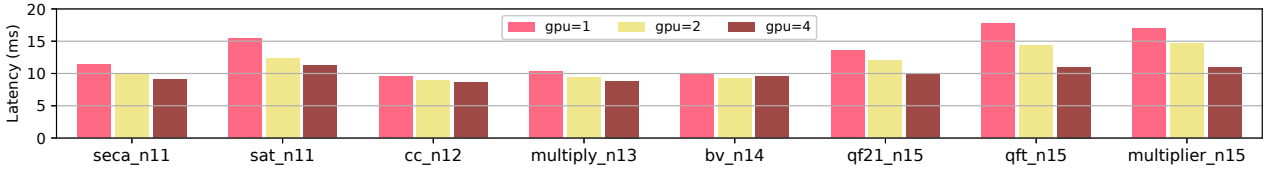

Figure 11: SV-Sim scaling-up evaluation on AMD MI-100 GPU workstation sing direct GPU peer-access.

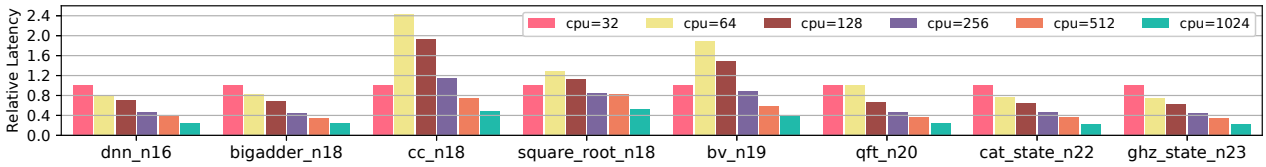

Figure 12: SV-Sim scaling-out evaluation on Summit HPC Power-9 CPUs using OpenSHMEM.

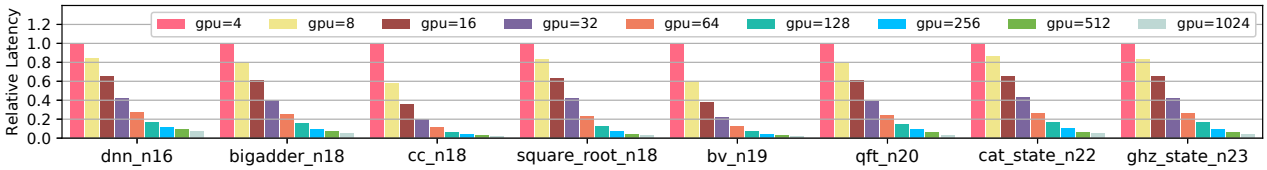

Figure 13: SV-Sim scaling-out evaluation on Summit HPC V100 GPUs using NVSHMEM.

communication bandwidth is only increased marginally with more nodes involved.

GPU NVSHMEM: We test the same 8 large-category circuit on the GPUs in the Summit supercomputer, scaling from 4 GPUs of a single node, to 1024 GPUs of 171-nodes, as each Summit node features 6 NVIDIA Volta V100 GPUs. The results are shown in Figure 13. Different from the OpenSHMEM-based CPU scenario, the NVSHMEM-based GPUs demonstrate strong scaling with the increasing number of GPUs. Again, the limitation here is not the computation kernel, but the network bandwidth for intensive SHMEM communication in the InfiniBand fabric.

Regarding AMD GPUs, since ROC_SHMEM is only a prototype software, and has very strict software dependency (i.e., ROCm-V2.10, ROCm-aware MPI via customized building, UCX-V1.6 with ROCm support) and hardware platform requirements (i.e., AMD GFX9 GPUs and ROCm-RDMA compatible InfiniBand adaptor), which our platform system cannot satisfy. Therefore, although we develop the ROC_SHMEM implementation, we are not able to test it in the evaluation. Enquiries with ROC_SHMEM developers confirmed full support of MI100 and more recent rocm runtime in future release of ROC_SHMEM.

4.4 Performance Comparison

Figure 14 compares the simulation latency of SV-Sim (CPU, CPU with AVX512, and V100 GPU) to the default simulators of the three widely used quantum programming environments: Qiskit (V0.26.2) from IBM, Cirq (V0.11.0) from Google, and Q# (V0.17.2105.143879) from Microsoft on the V100-DGX-2 platform (see Table 3) using a single CPU core or a single GPU. As can be seen, SV-Sim achieves significantly better performance (~10X on average) than these state-of-the-art state-vector quantum simulators, demonstrating the usefulness of the tool to the wide quantum computing community.

5 DISCUSSION

We discuss why SV-Sim is particularly useful for the development of variational quantum algorithms (VQAs) on the NISQ platforms.

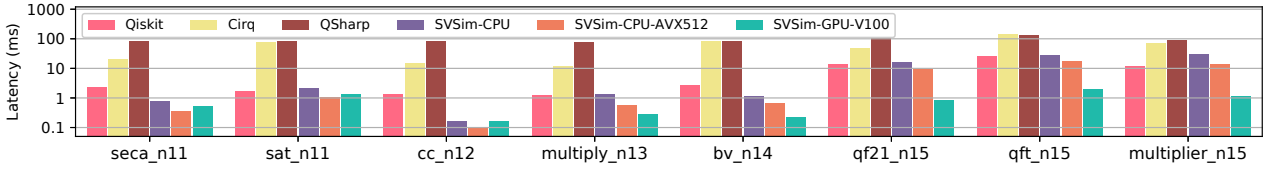

Figure 14: Simulation performance comparison.

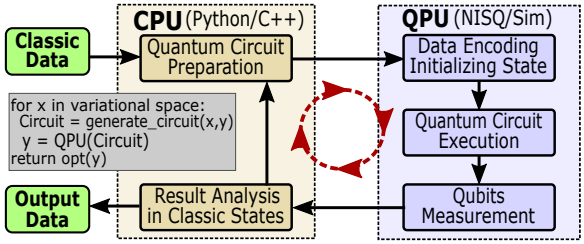

Figure 15: Variational quantum algorithms running on a quantum-classical hybrid architecture.

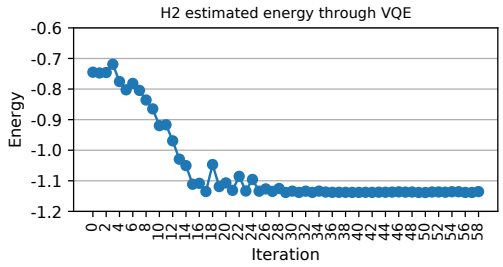

Figure 16: Estimated energy through VQE for H2.

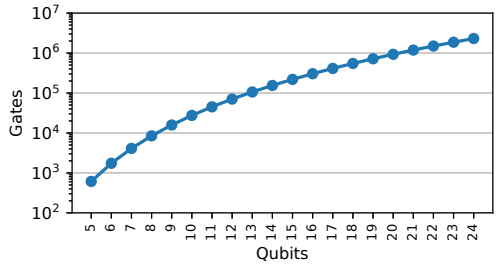

Figure 17: Gates in VQE with respect to qubits.

Figure 15 shows a general VQA framework. In each iteration of the variational search, the circuit to be executed in the quantum device or simulator is dynamically generated based on previous iteration results, with respect to certain searching strategies. Since most quantum programming environments (e.g., Qiskit, Cirq, Quil, ProjectQ, etc.) are based on Python or other high-level languages, in order to run these dynamically synthesized circuits in a single shot for high performance, without the proposed polymorphism design, one has to either parse the gates in the kernel (with branching overhead), or embrace JIT (by sustain compilation latency in the path). SV-Sim provides a very effective way to flexibly compose a kernel instance for an objective functionality. This is not only useful for quantum circuit simulation, but a wide range of domains such as deep learning [30, 40]. In the following, we discuss the application of SV-Sim on two practical use-cases: QNN and VQE.

QNN for Power-Grid: In this application, we investigate the utilization of quantum machine-learning in the power-grid domain. We develop a prototype variational quantum neural network (QNN) similar to Figure 1 for predicting the violation of contingency for an IEEE 30-bus system. We use the generator real power, generator reactive power, real load and reactive load values as the inputs. The QNN model outputs the probability of violation for unknown test cases. As a prototype demonstration, we train a small QNN using a dataset of 20 contingency cases. Each trial circuit has dozens of controlled rotational gates. After two epochs, the testing accuracy increases from 28.11% to 72.97%. Comparatively, a 4-layer 256-neuron classical MLP DNN reports a test accuracy increase from 62.87% to 74.05%, with 120 epochs. We use SV-Sim via the Python-API in Tensorflow to perform the training. We find that, only for this small problem, in one epoch training, the circuit has to be dynamically adjusted and verified for 28,641 times. Using SV-Sim, each trial takes merely ~0.6 ms thanks to the unique optimizations, much shorter than alternative simulators. The overall training can be finished in around 20 minutes. We can imagine for a large-scale training, e.g., on a 300-bus with thousands of contingency training data, what would be the time cost. Without an efficient simulator such as SV-Sim, the algorithm development for preparing such practical quantum applications would be difficult and quite time-consuming.

VQE for Chemistry: In this application, we evaluate the *Unitary Coupled Cluster Single-Double* VQE ansatz [57] for estimating the energy of the bound in the H2 molecule from QDK. Following the approach we discussed, the Q# code is translated into QIR. Operations defined in QIR are handled by the QIR-runtime, which embeds an SV-Sim instance for actual execution. Figure 16 shows the estimated energy with 58 iterations using the *Nelder-Mead* optimizer. The circuit uses 10 qubits and comprises ~86 gates. With SV-Sim, each circuit validation only takes 1.23ms using a V100 GPU on Summit HPC.

With more complex molecules, the number of gates increases significantly. Figure 17 shows the volume of gates with respect to the number of qubits using the VQE code from Scaffold [2]. From 5 qubits to 24 qubits, the number of gates that has to be evaluated in each VQE iteration increases from six hundreds of gates to 2.3 millions of gates. This level of depth is far beyond what existing simulators such as Qiskit have showcased before, as well as the coherent duration an NISQ machine can sustain. Using the 16 GPUs in a V100 DGX-2 system, SV-Sim is able to accomplish one such simulation trial in 196s or 3.5 mins, demonstrating the usefulness of SV-Sim for simulating ultra deep variational quantum circuits. Future work will seek further SV-Sim performance improvement for VQA through batched simulation, tensorcore acceleration [40], and alternative GPU-specific optimizations [33, 34, 37–39, 42, 43].

6 RELATED WORK

There have been many quantum simulators reported ever since the concept of QC first proposed in the 90s, some of which are summarized in the QC simulator zoo [1]. We briefly survey some of the latest ones.

Regarding CPU-based distributive quantum circuit simulation, Haner et al. [23] presented a CPU-based distributive quantum emulator that instead of simulating the elementary gate sequences, emulated quantum subroutines at a higher abstraction of mathematical description. Their customized emulation approach of repeated squaring U, being applicable to certain quantum routines (e.g., FFT and QPE), could considerably reduce computing complexity. Later, Haner and Steiger [22] reported the simulation of a 45-qubit circuit on NERSC Cori HPC. They still used MPI but introduced an extra clustering layer to facilitate circuit reform and reduced the amount of communication. Wu et al. [59] applied lossy data-compression technique in order to sustain more qubits into the memory. They were able to simulate several more qubits but at the cost of introducing extra simulation error. They tested on ANL Theta HPC using MPI. Their compression technique is perpendicular to SV-Sim. De Raedt et al. [15] revised and extended the Fortran-based JUQCS simulator [16] and tested it on several HPCs including K-computer and Sunway TaihuLight. Still, their communication approach is the same as [16], which is to swap local qubits with remote qubits by tracking and updating the permutation of the qubit indices. JUQCS was only for CPU clusters. Pang et al. [49] accelerated the contraction and refactorization for tensor-network based quantum simulation. Jones et al. presented QuEST [27], a CPU-based distributed state-vector quantum simulator relying on traditional MPI-based communication. They also reported a CUDA implementation, but was only for a single GPU. QuEST did not have a Python interface and was validated using a randomly generated circuit. In the TensorFlow-Quantum [11], a C++ simulator called qsim was proposed. The major optimization performed is gate fusion and vectorization through AVX2. However, qsim is only for a CPU in a single node.

Regarding multi-GPU based simulation, Zhang et al. [60] presented a state-vector simulator for a single GPU-node comprising 4 NVIDIA Kepler K20 GPUs. They exploited inter-gate locality to reduce the amount of communication. However, their communication is performed through the host-side system memory. Intensive communication burden is imposed over the PCI-e bus between the CPUs and GPUs. Li and Yuan [44] observed the PCI-e limitation, and proposed a state-vector simulator for a small-scale GPU cluster (4 nodes each with 4 NVIDIA K20 GPUs). To mitigate communication overhead, they presented a new data distribution method to exploit data locality. However, the communication is still performed on the CPU-side with per-gate host-device communication. Doi et al. [17] proposed a state-vector quantum simulator for multi-nodes multi-GPUs. They tried to leverage the strong computation capability of GPUs but the large memory of CPUs. They partitioned the state-vector into chunks and managed the swapping of chunks between CPU and GPUs, for each gate simulation. They tested on a GPU cluster with 32 nodes with each node having the similar architecture as Summit (6 NVIDIA V100 GPUs). Finally, our previous work [41] presented a density matrix quantum circuit simulator for GPU clusters. To gain better performance, all the gates of a circuit were fused into a single GPU kernel. This might lead to long compilation time and if the circuit is dynamically synthesized such as in a variational algorithm, the compilation overhead can be a major concern. Additionally, the communication pattern for density matrix simulation is quite different from state-vector. The optimization applied for the host-side MPI-based communication cannot be adopted here for state-vectors.

All of the existing works, however, rely on CPU-side MPI-based communication for distributed simulation, which lead to considerable communication and synchronization overhead for CPU-device data exchange. In order to improve communication granularity, a runtime data coordination scheduler is often required, leading to considerable overhead and missing the fine-grained computation/-communication overlapping opportunities in state-vector quantum simulation. SV-Sim is the first work to leverage advanced PGAS-based SHMEM communication models such as GPUDirect-Peer, OpenSHMEM and NVSHMEM, for tackling the massive irregular fine-grained communication. SV-Sim is also the first work to offer both C++ and Python level interface for large-scale simulation across all major CPUs and accelerator backends (i.e., NVIDIA GPUs, AMD GPUs and Intel MIC).

7 CONCLUSION

Observing the major obstacle for scalable state-vector quantum circuit simulation in classical HPC is the massive fine-grained communication, in this paper we present SV-Sim — a novel state-vector simulator leverages the PGAS-based communication model (direct access for intra-node and SHMEM for inter-node) for high-performance simulation across a variety of modern HPC platforms, in a unified framework. Evaluations over six platforms including A100-DGX-1, V100-DGX-2, OLCF Summit and ALCF Theta demonstrate the outstanding performance and scalability of SV-Sim. SV-Sim serves as a useful validation and verification tool for variational quantum algorithm development in the NISQ era. Future work includes building a variational algorithm specific simulator by further parallelizing the variational optimization loop and applying alternative advanced optimizations.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Quantum Science Center. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

REFERENCES

[1] [n.d.]. List of QC simulators. https://www.quantiki.org/wiki/list-qc-simulators.
[2] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum programming language.* Technical Report. Princeton Univ NJ Dept of Computer Science.
[3] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, D Bucher, FJ Cabrera-Hernández, J Carballo-Franquis, A Chen, CF Chen, et al. 2019. Qiskit: An open-source framework for quantum computing. *Accessed on: Mar* 16 (2019).
[4] AMD. [n.d.]. ROCm OpenSHMEM. URL: https://github.com/ROCm-Developer-Tools/ROC_SHMEM.
[5] AMD. 2020. AMD Infinity Fabric.

[6] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[7] Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical review A* 52, 5 (1995), 3457.

[8] Rami Barends, Julian Kelly, Anthony Megrant, Andrzej Veitia, Daniel Sank, Evan Jeffrey, Ted C White, Josh Mutus, Austin G Fowler, Brooks Campbell, et al. 2014. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature* 508, 7497 (2014), 500–503.

[9] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. 2020. Training deep quantum neural networks. *Nature communications* 11, 1 (2020), 1–6.

[10] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. 2018. Characterizing quantum supremacy in near-term devices. *Nature Physics* 14, 6 (2018), 595–600.

[11] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergei V Isakov, Philip Massey, Murphy Yuezhen Niu, Ramin Halavati, Evan Peters, et al. 2020. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989* (2020).

[12] Jonathan Carter, David Dean, Greg Hebner, Jungsang Kim, Andrew Landahl, Peter Maunz, Raphael Pooser, Irfan Siddiqi, and Jeffrey Vetter. 2017. *ASCR Report on a Quantum Computing Testbed for Science.* Technical Report. USDOE Office of Science (SC), Washington, DC (United States). Advanced ....

[13] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 1–3.

[14] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017). Repo: https://github.com/Qiskit/openqasm.

[15] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Willsch, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. 2019. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications* 237 (2019), 47–61.

[16] Koen De Raedt, Kristel Michielsen, Hans De Raedt, Binh Trieu, Guido Arnold, Marcus Richter, Th Lippert, Hiroshi Watanabe, and Nobuyasu Ito. 2007. Massively parallel quantum computer simulator. *Computer Physics Communications* 176, 2 (2007), 121–136.

[17] Jun Doi, Hitomi Takahashi, Rudy Raymond, Takashi Imamichi, and Hiroshi Horii. 2019. Quantum computing simulator on a heterogeneous hpc system. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 85–93.

[18] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

[19] Edward Farhi and Hartmut Neven. 2018. Classification with quantum neural networks on near term processors. *arXiv preprint arXiv:1802.06002* (2018).

[20] Gian Giacomo Guerreschi and Anne Y Matsuura. 2019. QAOA for Max-Cut requires hundreds of qubits for quantum speed-up. *Scientific reports* 9, 1 (2019), 1–7.

[21] Khaled Hamidouche and Michael LeBeane. 2020. Gpu initiated openshmem: correct and efficient intra-kernel networking for dgpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 336–347.

[22] Thomas Häner and Damian S Steiger. 2017. 5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10.

[23] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 866–874.

[24] Bettina Heim. 2021. Universal Quantum Intermediate Representation. *Bulletin of the American Physical Society* (2021).

[25] IBM. [n.d.]. IBM Quantum Experience. URL: https://quantum-computing.ibm.com/.

[26] Sylvain Jeaugey. 2017. NCCL 2.0. In *GPU Technology Conference (GTC)*.

[27] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific reports* 9, 1 (2019), 1–11.

[28] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549, 7671 (2017), 242–246.

[29] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K Reinhardt, and Lizy K John. 2017. GPU triggered networking for intra-kernel communications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[30] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. 2019. BSTC: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–30.

[31] Ang Li and Sriram Krishnamoorthy. 2020. QASMBench: A low-level QASM benchmark suite for NISQ evaluation and simulation. *arXiv preprint arXiv:2005.13018* (2020).

[32] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[33] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-consolidation: A novel execution model for gpus. In *Proceedings of the 2018 International Conference on Supercomputing*. 53–64.

[34] Ang Li, Shuaiwen Leon Song, Eric Brugel, Akash Kumar, Daniel Chavarria-Miranda, and Henk Corporaal. 2016. X: A comprehensive analytic model for parallel machines. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 242–252.

[35] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.

[36] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 191–202.

[37] Ang Li, Shuaiwen Leon Song, Akash Kumar, Eddy Z Zhang, Daniel Chavarría-Miranda, and Henk Corporaal. 2016. Critical points based register-concurrency autotuning for GPUs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1273–1278.

[38] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware CTA clustering for modern GPUs. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 297–311.

[39] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-driven transparent approximation acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–14.

[40] Ang Li and Simon Su. 2020. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1878–1891.

[41] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[42] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. 2015. Fine-grained synchronizations and dataflow programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 109–118.

[43] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[44] Zhen Li and Jiabin Yuan. 2017. Quantum computer simulation on gpu cluster incorporating data locality. In *International Conference on Cloud Computing and Security*. Springer, 85–97.

[45] Daniel A Lidar and Todd A Brun. 2013. *Quantum error correction.* Cambridge university press.

[46] A Linn. [n.d.]. The future is quantum: Microsoft releases free preview of quantum development kit.(Dec. 11, 2017).

[47] John Nickolls and William J Dally. 2010. The GPU computing era. *IEEE micro* 30, 2 (2010), 56–69.

[48] NVIDIA. [n.d.]. NVIDIA NVSHMEM Developer Guide. URL: https://docs.nvidia.com/hpc-sdk/nvshmem/archives/nvshmem-101/developer-guide/index.html.

[49] Yuchen Pang, Tianyi Hao, Annika Dugad, Yiqing Zhou, and Edgar Solomonik. 2020. Efficient 2D tensor network simulation of quantum systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[50] Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Manjunath Gorentla Venkata, Oscar Hernandez, Pavel Shamis, M Graham Lopez, Mathew Baker, and Wendy Poole. 2014. Exploring OpenSHMEM model to program GPU-based extreme-scale systems. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 18–35.

[51] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[52] Jonathan Romero, Ryan Babbush, Jarrod R McClean, Cornelius Hempel, Peter J Love, and Alán Aspuru-Guzik. 2018. Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz. *Quantum Science and Technology* 4, 1 (2018), 014008.

[53] Davide Rossetti and S Team. 2015. GPUDIRECT: Integrating the GPU with a Network Interface. In *GPU Technology Conference*.

[54] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[55] SPI. [n.d.]. OpenMPI: Open Source High Performance Computing. URL: https://www.open-mpi.org/.

[56] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. 1–10.

[57] James D Whitfield, Jacob Biamonte, and Alán Aspuru-Guzik. 2011. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics* 109, 5 (2011), 735–750.

[58] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[59] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–24.

[60] Pei Zhang, Jiabin Yuan, and Xiangwen Lu. 2015. Quantum computer simulation on multi-GPU incorporating data locality. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 241–256.