Density Matrix Quantum Circuit Simulation via the BSP Machine on Modern GPU Clusters

Ang Li*, Omer Subasi*, Xiu Yang*† and Sriram Krishnamoorthy*‡
*Pacific Northwest National Laboratory (PNNL), Richland, WA, 99354, USA
†Lehigh University, Bethlehem, PA, 18015, USA
‡Washington State University, Pullman, WA, 99164, USA
Email: ang.li@pnnl.gov, omer.subasi@pnnl.gov, xiy518@lehigh.edu, sriram@pnnl.gov

***Abstract*—As quantum computers evolve, simulations of quantum programs on classical computers will be essential in validating quantum algorithms, understanding the effect of system noise, and designing applications for future quantum computers. In this paper, we first propose a new multi-GPU programming methodology called MG-BSP which constructs a virtual BSP machine on top of modern multi-GPU platforms, and apply this methodology to build a multi-GPU density matrix quantum simulator called DM-Sim. We propose a new formulation that can significantly reduce communication overhead, and show that this formula transformation can conserve the semantics despite noise being introduced. We build the tool-chain for the simulator to run open standard quantum assembly code, execute synthesized quantum circuits, and perform ultra-deep and large-scale simulations. We evaluated DM-Sim on several state-of-the-art multi-GPU platforms including NVIDIA's Pascal/Volta DGX-1, DGX-2, and ORNL's Summit supercomputer. In particular, we have demonstrated the simulation of one million general gates in 94 minutes on DGX-2, far deeper circuits than has been demonstrated in prior works. Our simulator is more than 10x faster with respect to the corresponding state-vector quantum simulators on GPUs and other platforms. The DM-Sim simulator is released at: *http://github.com/pnnl/DM-Sim*.**

I. Introduction

Despite holding substantial promise, quantum computing (QC) based on today's *noisy-intermediate-scale-quantum* devices (NISQ) [72] is still distant from beating classical supercomputers. One major limitation is the error, particularly the decoherence of qubits which introduces significant uncertainty to the quantum states and corrupts the functionality of the quantum circuit. The stable coherence duration varies across different QC technologies, from microseconds to seconds with different error rate [2] and readout fidelity [3]. Consequently, identifying how the introduced error propagates among qubits and along the circuit becomes a critical issue for QC research.

Directly inspecting the intermediate states of a physical quantum computer is, however, infeasible. Due to fundamental quantum rules, whenever a measurement is applied to certain qubits, it destroys the superposition state and alters the computation logic. Consequently, simulating the quantum circuit (see Figure 1) through classical computers becomes a necessary approach to unfold the black-box, investigate the error, and validate the quantum algorithm and hardware in a more tractable approach. This is particularly the case when theoretical bounds are inherently imprecise (e.g., Trotter error bounds for time evolution of a Hamiltonian [8]). Furthermore, efficient classical simulations can also form the starting point for quantum-inspired algorithms where an algorithm derived for QC is reversely deployed on a classical computer [59].

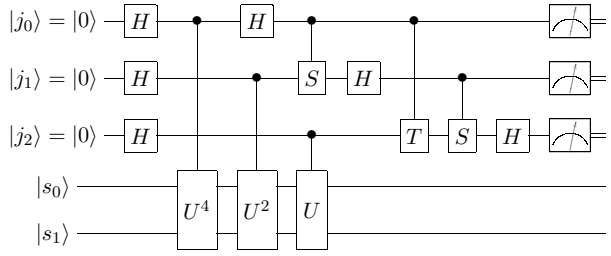

Fig. 1: Quantum circuit. The horizontal lines represent qubits. The blocks along the lines represent gates. Execution is from left to right.

A variety of quantum circuit simulators on classical computers have been proposed [1]. However, most of them target logical qubits in an ideal isolated environment where full gate fidelity can be asserted, rather than physical qubits in a practical open environment with unavoidable noise. Since QC simulation demands huge amount of memory [11], these simulators often rely on state vector $|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$ to conserve the *pure* quantum states. The transformation by applying a quantum gate described as an unitary operator U thus is $|\psi\rangle \rightarrow U|\psi\rangle$. However, when noise is introduced, we have to deal with a *mixed* state comprising a statistical ensemble of multiple distinct quantum systems in a *density matrix*. In an ensemble, if the M different quantum systems are in states $|\psi_j\rangle$ with probability P_j ($1 \leq j \leq M$), the density matrix can be expressed as $\rho = \sum_{j=1}^{M} P_j |\psi_j\rangle \langle\psi_j|$. However, P_j in a real quantum physical system is typically statistical rather than determinable. Sometimes it is unknown on purpose [9].

There are two fundamental types of quantum gate errors: (i) *Coherent errors* conserve the purity of the input state. Instead of executing U, another unitary operation $\tilde{U}$ is essentially applied. (ii) *Incoherent errors* do not conserve the purity of the input state. As a result, the state transition can only be described through a density matrix [65], which contains all information necessary to decide the probability of any outcome of the circuit in any future measurement. Therefore, to simulate quantum circuits for NISQ devices lacking quantum error correction (QEC) support, being able to simulate using density matrix can be crucial, sometimes inevitable [9], [35].

The major difference between density matrix and state vector simulation is that: to simulate the same amount of qubits n, the memory access and occupation, the computation,

SC20, November 9-19, 2020, Is Everywhere We Are

the network data exchange all scale in $O(4^n)$ rather than $O(2^n)$, leading to great pressure on the hardware resources consumption and dramatically expanded simulation time. In fact, existing work simulates n-qubits density-matrix through $2n$-qubits state vector [33]. Due to the different simulation purposes, in this work we do not seek to simulate more qubits than state-of-the-art state-vector simulators [13], [68], [84] through techniques such as using secondary storage [68], presuming particular gates [13], and lossy compression [84]. Rather, this work attempts to tackle three major challenges: **(i) Circuit depth:** As a well-known approach, quantum simulators tend to decompose each multi-qubits gate into a sequence of 1-qubit or 2-qubits gates [4], [6], [7], [15], [22]–[24], [26], [27], [34], the actual circuit for useful quantum algorithms can be extremely deep, such as quantum chemistry simulation (e.g., 10^{14} gates [73], 10^{18} gates [65]), quantum approximate optimization algorithm (QAOA) [17], and quantum neural networks [18]. This is exacerbated when hardware constraints are taken into consideration [51]. However, existing simulators often simulate from a single gate to a few hundreds of gates [15], [22], [24], [26], [34], [68]. **(ii) Performance:** Due to the large problem size, density matrix simulation can be remarkably slow. This condition can be even worse with very deep circuits and the fact that simulations are often repeated many times to, for example, obtain converged distribution sampling, study the influence of hyper-parameters and noise, and train a quantum machine learning model, etc. **(iii) Programming flexibility:** Although the major target is to support arbitrary density matrices for error study, defining new gates with advanced optimization should be straightforward. In other words, the simulator should ensure the programmability of the quantum gates while conserving execution efficiency.

Due to massive fine-grained parallelism, intensive double-precision compute, scarce memory bandwidth demand and exhaustive inter-processor communication, state-of-the-art multi-GPU HPC systems become the ideal platforms for density matrix quantum simulation, particularly concerning GPUs' massive lightweight parallel threads, excellent double-precision floating-point capability, high-bandwidth device memory (e.g., HBM, GDDR) and high-speed interconnect (e.g., NVLink[1]).

However, designing an efficient multi-GPU density matrix quantum circuit simulator still remains challenging because (I) A *programming methodology* on how to manage the massive threads for efficient inner-GPU computation and inter-GPU cooperation is still missing. This may partially explains why all existing density-matrix quantum simulators (and most state vector simulators) are using a single GPU [4], [6], [7], [22]–[24], [27], [33], [34], [76]; (II) Per-gate *global communication and synchronization*. Unlike state vector gate operation $|\psi\rangle \rightarrow U|\psi\rangle$, applying density matrix gate $\rho \rightarrow U\rho U^\dagger$ demands adjoint operation $U^\dagger = (U^*)^T$, which involves global transposition, implying all-to-all data exchange which can incur tremendous communication overhead. Additionally, global synchronization is required per gate due to data dependency while the processing of a single gate is in a streaming manner with rare data reuse. Traditionally, GPUs lack whole-device inter-thread-block synchronization mechanism and GPU-side-initiated communication mechanism. As a result, existing designs tend to simulate one gate per GPU kernel followed by CPU-side communication or synchronization, introducing considerable overhead from repeated kernel invocation & release [22] (GPU kernel start latency can be as long as $20\mu s$ [36]), and frequent CPU-GPU execution transition. This overhead is further amplified with multi-GPUs and multi-nodes; (III) *Interconnect*. Despite this being the first simulator using multi-GPUs for density matrix simulation, an existing work has already applied multi-GPUs for state vector simulation [86]. However, it did not leverage the recent advancement in GPU interconnect, partially due to the CPU-centric programming model where all GPUs are managed by a master CPU and data exchange is only between CPU and GPUs; (IV) *Optimization*. Traditional simulators tend to precisely reflect particular QC technology parameters [66], rather than formulating the process from an efficient simulation angle, leading to redundant computation and communication.

In this paper, we propose a new programming methodology called *MG-BSP* that constructs a virtual *Bulk-Synchronous-Parallel* machine on top of modern multi-GPU platforms. We demonstrate its programmability and efficiency by applying it to the density matrix quantum circuit simulation problem, and evaluate the performance on several multi-GPU platforms, including NVIDIA DGX-1 [61], DGX-2 [63] and ORNL Summit HPC. The results demonstrate the effectiveness, flexibility and scalability of MG-BSP and the DM-Sim simulator. Particularly, we show that a 15-qubit density matrix simulation (i.e., the same scale as a 30-qubit state vector simulation) with **1 million** arbitrary gates can be accomplished in 94 minutes ($\approx$5.6 ms/gate) on the NVIDIA DGX-2 system with 16 Volta V100 GPUs, far deeper and quicker than has been demonstrated before. In summary, this paper makes the following contributions:

- We propose a novel multi-GPU programming methodology called MG-BSP which constructs a virtual BSP-machine on top of modern multi-GPU devices. The programming model offers good performance, programmability, and portability, which can be further tweaked for other utilizations and other platforms.
- Based on MG-BSP, we build a density matrix quantum circuit simulator DM-Sim for multi-GPUs. It fills the gap of lacking a quantum circuit simulator specially for density matrix accelerated by multi-GPUs. The simulator is able to run through the entire quantum circuit using a single or dual GPU kernel(s), posting significantly better performance than corresponding state-of-the-art state vector simulators. We build the tool-chain for supporting OpenQASM assembly code in our simulator.
- We propose a new formula transformation approach for density matrix quantum circuit simulation, which significantly reduces the communication overhead for deep circuits. We show that such a transformation conserves the semantics of error with noisy quantum gates.

[1]In this paper, we use NVIDIA and CUDA terminology for convenience as our platforms are NVIDIA GPUs. However, the proposed techniques can be applied to other vendors' GPUs as well.

II. Virtual BSP Machine on Multi-GPU

A. MG-BSP Programming Methodology

This new Multi-GPU programming model is labeled *MG-BSP*, which originates from the *Bulk-Synchronous-Parallel* model [19], [79]. The basic BSP model comprises three parts: a number of components that can perform computation and memory functions; a router that delivers messages point-to-point among components; and a facility for global synchronization among all components. Please refer to [12], [19], [79] for more details, as the BSP model is classic and well-known. Regarding multi-GPU scenario, from threads to GPU kernels, four levels of BSP model can be constructed from bottom up:

- **Threads** or lanes in a warp can perform computation on various function units (e.g., scalar units, special function units, tensor cores), and register access. They communicate through specific voting instructions (e.g., `__ballot()`, `__all()`, `__any()`) and register-shuffling. They synchronize through warp level barrier `__syncwarp()`.
- **Warps** in a thread block can perform computation on various function units and memory access on various memory storage (e.g., shared memory, constant memory, global memory). They communicate through shared memory [49] or global memory [45] while synchronizing via thread-block level barrier primitive "`__syncthreads()`".
- **Thread Blocks** in a GPU kernel can perform computation or memory operations on different SMs. They communicate via L2 cache or global memory. Traditionally, GPU lacks thread block level synchronization mechanism — a new GPU kernel is usually invoked to enforce an implicit barrier among thread blocks.
- **Thread Grids** or GPU kernels perform computation and memory operations on different GPUs. They communicate via point-to-point interconnects such as PCI-e, NVLink, and NVSwitch. They synchronize through barriers from a higher level context such as OpenMP (GPUs managed by OpenMP threads) and MPI (GPUs managed by MPI processes).

The MG-BSP model merges the four BSP levels into a single BSP virtual machine. We view all threads from multi-GPUs as uniform. They communicate with each other through inter-GPU point-to-point access in a memory access superstep. They synchronize through the new global synchronization method "`grid.sync()`". MG-BSP has the following major features:

- **Single kernel**: throughout the entire application execution, only one GPU kernel is invoked. In other words, all GPU functions are fused into a single kernel with the same configuration. This entirely eliminates the overhead from repeated kernel invocation, initialization and release.
- **Atomic superstep:** MG-BSP assumes unavoidable data dependency among supersteps, so each step is independent and should be executed atomically. This is achieved by probing a global barrier `grid.sync()` beneath each superstep.
- **GPU-driven:** Given the current trends on GPU file-system [74], RDMA [70] and GPU-initiated communication [71], it is foreseen that future GPU-based HPC cluster would potentially connect InfiniBand or other inter-node network directly to GPUs (rather than via PCIe). In that sense, GPUs are no longer the accelerators managed by CPUs but the major computing and communication processors of an HPC cluster. MG-BSP is well-suited for this GPU-centric model.

```
1 #include <cooperative_groups.h>
2 using namespace cooperative_groups;
3 //Define ISA Format Header
4 #define ISA_FORMAT_HEADER tid=blockDim.x*blockIdx.x + \
5  threadIdx.x; for(int i=tid;i<Size;i+=blockDim.x*gridDim.x){
6 //Define ISA Format Footer
7 #define ISA_FORMAT_FOOTER } grid.sync();//Sync per superstep
8 //Define a sample MG-BSP instruction
9 __inline__ __device__ Instruction_1(X_out, X_in, C_param){
10   ISA_FORMAT_HEADER
11     ...(X_out, X_in);//processing
12   ISA_FORMAT_FOOTER;}
13 __device__ Subprogram_1(X_out, X_in){ //Define a subprogram
14     Instruction_2(X1,X_in,C1);
15     Instruction_50(X2,X1,C2);
16     ...}
17 __global__ void Program(T* X0){ //Define MG-BSP Program
18     grid_group grid = this_grid();//for global barrier
19     //instr(out, in, param)
20     Instruction_3(X1,X0,C1);//F1 is an instance of f3
21     Instruction_7(X2,X1,C2);//F2 is an instance of f7
22     Instruction_4(X3,X2,C3);//F3 is an instance of f4
23     Instruction_6(X4,X3,C4);//F4 is an instance of f6
24     Instruction_6(X5,X4,C5);//F5 is another instance of f6
25     ...
26     //subprogram blocks
27     Subprogram_6(Xb1, Xb0);//Blk-1
28     Subprogram_9(Xb2, Xb1);//Blk-2
29     Subprogram_1(Xb3, Xb2);//Blk-3
30     ...
31     //other user-defined routines
32     P2P_Communication(X_out, X_in);
33     Compression(X_in);
34     Decompression(X_out);
35     ...}
36 //MG-BSP Machine Configuration
37 cudaDeviceProp deviceProp; int numBlocksPerSm;
38 cudaGetDeviceProperties(&deviceProp, dev);
39 cudaOccupancyMaxActiveBlocksPerMultiprocessor(
40     &numBlocksPerSm, simulation, THREADS_PER_BLOCK, 0);
41 dim3 gridDim(numBlocksPerSm*deviceProp.multiProcessorCount);
42 void* args[]={&X0};
43 //Execute the defined Program
44 cudaLaunchCooperativeKernel((void*)Program, gridDim,
45         THREADS_PER_BLOCK, args, 0);
```

Listing 1: MG-BSP Virtual Machine.

We label MG-BSP as a virtual machine because it offers flexible programming abstraction. It consecutively executes a series of m atomic operators F over the input data X_0 through:

$$X_i = \begin{cases} X_0 & \text{if } i = 0 \\ F_i(X_{i-1}, C_i) & F_i \in \Omega\{f_0, \cdots f_{n-1}\},\ 0 < i < m \end{cases} \tag{1}$$

Each F is a computation or communication operator executed in a superstep. F_i is an instance of instruction f_j, with $0 \le j < n$ and parameter C_i, from the MG-BSP *ISA* — $\Omega\{f_0, \cdots f_{n-1}\}$ including n instructions. The MG-BSP virtual machine is to execute a program comprising a sequence of instructions: $P = F_{m-1}F_{m-2}\cdots F_0$. The user defines the ISA Ω and program P, while the virtual machine executes P according to Ω.

Listing 1 illustrates our MG-BSP programming skeleton. The ISA (i.e., Ω) comprises a series of inlined GPU device functions. A sample is given in Line-9 to 12. Each device function concretizes a virtual instruction f_j. All instructions are defined under a uniform thread configuration and potentially a uniform accessing pattern over the data stream X. To enforce uniform thread configuration, we need: (a) *the*

same dimension and number of threads per thread block (i.e., *THREADS_PER_BLOCK* in Listing 1). This can be achieved via thread coarsening [58], [80], [85] or warp consolidation [40]. Based on our experience, 32, 256, and 1024 are the most appropriate values. Using 32 is to allocate as many fine-grained thread blocks as possible to avoid false waiting at the warp level [40], and potentially benefit from cache bypassing [50] and warp throttling [41], [46]. 256 is the default configuration suggested by *CUDA Best Practice Guide* [62] for attaining optimal occupancy [44], [48]. 1024 is adopted when data sharing within a thread block is particularly crucial to performance, such as sharing a square tile of 32×32 [49]. (b) *the same dimension and number of thread blocks per kernel*. This can be achieved using a task model that traverses the entire thread block space (i.e., all thread blocks of the kernel) with certain amount of elastic thread blocks that can just fully leverage all the SM warp slots, known as elastic kernel [67] or warp delegation [45]. Line 4-5 define the header to ensure all thread block jobs will be executed. Line 37-41 extract the number of thread blocks that can achieve the best occupancy under the present *THREADS_PER_BLOCK* value. (3) *the same dynamic shared memory allocation per thread block*. This depends on individual algorithm design. Finally, to ensure data dependency and instruction atomicity, the virtual machine globally synchronizes at Line 7.

We then discuss the Program (i.e., *P*), which is defined as a sequence of instruction calls encapsulated in a global function (Line 17 in Listing 1). *P* may include subprogram blocks for two reasons: (i) *Programmability*: when *P* becomes increasingly complex, we need hierarchical abstraction levels. The call graph of GPU device functions is eventually handled by the GPU compiler, which offers an ideal approach to resolve the abstraction dependency and semantics, as will be shown later. (ii) *Scalability*: when *P* becomes even more complex, manually generating *P* is difficult and we have to rely on automatic assembly tools such as a DSL compiler or a script generator to synthesis the instruction sequences. When that happens, the synthesized results can be encapsulated as subprograms (e.g., Line 13-16). To offer additional flexibility, these subprograms can be saved as independent "*.cuh*" header files, and later invoked by predefined APIs inside *P*. Finally, the programmer can define other helper functions, such as the multi-GPU communication function, data compression, etc. to be leveraged and assembled in the program *P*.

B. MG-BSP Communication Model

So far we have only discussed BSP at the thread and thread block level of a single GPU. Now we transit to the multi-GPU and multi-GPU communication interface. In the past few years, the number of GPUs in a machine increases from one traditionally, to two in a *scalable-link-interface* (SLI) based system, to four in a GPU workstation, to eight in a DGX-1 system [61], and to 16 in a DGX-2 system [63]. In the meanwhile, interconnect among GPUs also evolved from conventional CPU-centric PCI-e tree network, to dual GPU SLI-bridge, to 3D-hypercube NVLink network in DGX-1, to all-to-all NVSwitch network in DGX-2. Figure 2 illustrates the interconnect topology of the five platforms we adopted for our evaluation (listed in Table II). Note that for DGX-1P and DGX-1V, we only show GPU-0 to GPU-3. More information on multi-GPU interconnect can be found in [42].

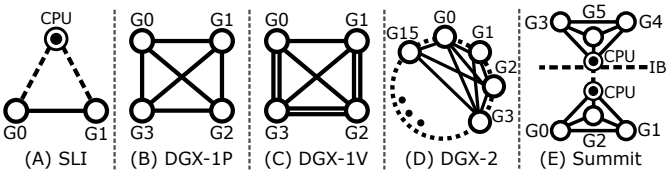

Fig. 2: Multi-GPU interconnect network topology.

```
1 //Unified memory for sharing among CPU and GPU
2 //Semaphores for GPU tell CPU on starting comm
3 __device__ __managed__ int start_comm[N_GPUS];
4 //Semaphores for CPU tell GPU on finishing comm
5 __device__ __managed__ int finish_comm[N_GPUS];
6 //GPU-side Communication Instruction
7 __device__ Instruction_Comm(...){
8   if (blockIdx.x == 0 && threadIdx.x==0){
9     //signal CPU thread to start P2P comm via GPU network
10    atomicExch(&start_comm[gid],1);
11    //wait for CPU's finish signal
12    while(atomicExch(&finish_comm[gid],0)!=1); }
13  grid.sync(); } // Other GPU threads to wait thread-0
14 //CPU-side P2P Async Communication
15 #pragma omp parallel num_threads (n_gpus)
16 {
17   int gid=omp_get_thread_num();
18   cudaSetDevice(gid);
19   cudaLaunchCooperativeKernel(...); //launch kernel
20   while (start_comm[gid]==0){}//wait for CPUs' start signal
21 #pragma omp barrier
22   start_comm[gid]=0;//reset
23   //Perofrm P2P communication
24   cudaMemcpyAsync(...);
25 #pragma omp barrier //ensure all CPU threads finish
26   finish_comm[gid]=1;//signal GPU on finishing comm
27 }
```

Listing 2: MG-BSP Communication Instruction with OpenMP.

Regarding the GPU-level BSP, we have each GPU as a processing component. For inter-GPU communication, there are several approaches, including traditional CPU forwarding, GPU peer access [64], unified memory [64], NCCL [60], NVSHMEM [71], and MPI with GPUDirect RDMA [70]. MG-BSP itself does not impose any preference or restrictions on a particular approach; the best choice depends on the communication patterns of the applications. Nevertheless, the CPU forwarding approach transfers data using CPU as the router, leaving the new inter-GPU interconnect unused at all. Besides, based on our tests, the unified memory generally shows inferior communication efficiency, as will be discussed. Therefore, for MG-BSP in a single node, we can use peer access, NCCL or NVSHMEM, depending on the applications. If the communication has to be initiated from the GPU-side without aborting the present kernel, NVSHMEM and a method proposed here are feasible; otherwise, from the CPU side, we can adopt NCCL for supported collective communication and peer access for flexible P2P communication. For MG-BSP among nodes, MPI, NCCL and NVSHMEM are all feasible.

Listing 2 illustrates our proposed approach where inter-GPU communication can be triggered from the GPU-side based on peer access. Each GPU is managed by an OpenMP thread. The synchronization is through the producer-consumer paradigm. The idea is to allocate spin locks on the unified memory [64] (e.g., Line 3, 5), where the CPU waits (Line 20) until the

lock has been released from the GPU-side. Then, the GPU waits at another lock (Line 12) until the CPU finishes the communication and releases the second lock (Line 22). Note that the spinning and release operations on the GPU-side has to be atomic. We have tested several designs, and this seems to be the only workable approach without incurring deadlocks.

Although Listing 2 runs correctly, the performance is essentially lower than simply terminating the present GPU kernel, doing the communication on the CPU-side and invoking a new GPU kernel to continue, which shows $\sim$6$\times$ better performance than Listing 2 in our tests. This is likely due to (1) GPU threads barrier in Line 13; (2) CPU threads barrier in Line 21 and 25; (3) repeated atomic access on slow unified memory in Line 12; and (4) CPU spinning access in Line 20. Within these, (3) is the major reason, since unified memory relies on page faulting mechanism to provide system-wide virtual space between CPUs and multi-GPUs [64].

Regarding multi-nodes scenario, we rely on MPI to manage the multi-GPUs of different nodes. The communication and synchronization are through MPI communication and synchronization primitives. When *GPUDirect-RDMA* [70] is enabled, no CPU-side buffer or involvement are required. The design in Listing 2 can be instantly adapted by replacing "`#pragma omp barrier`" with "`MPI_Barrier()`", and performing MPI communication accordingly under *GPUDirect-RDMA*.

C. MG-BSP Summary

To summarize, MG-BSP has the following advantages:

- **Performance:** the performance gains from (i) significantly reduced GPU function calls with all kernels being fused; (ii) as the thread hierarchy configurations (including thread block & grid dimensions and sizes, shared memory usage) are delegated to the virtual machine, an ideal configuration best fitting the underlying hardware can be specified by MG-BSP for always achieving optimal GPU SM occupancy. (iii) The elastic execution model [67] can trigger advanced scheduling and optimization opportunities [45], [83].
- **Programmability:** programmability is granted in that: (i) domain experts such as quantum physicists do not need to learn the details about GPU before applying MG-BSP to build a quantum simulator and attain superior performance; (ii) the task of finding the best kernel configuration is delegated to MG-BSP; (iii) since the user only provides the ISA and program, design modification is straightforward. We also provide methods for automatic program (P) generation.
- **Applicability:** MG-BSP abstracts the complex multi-GPU HPC platforms and delegates kernel settings, ensuring good applicability over different generations of NVIDIA GPUs and interconnects.

MG-BSP is most beneficial under the following scenarios: (I) the application can be efficiently executed using a BSP model, e.g., regular execution with balanced workload per superstep. (II) data dependency exists per superstep. (III) there is rare data reuse across supersteps (e.g., streaming access).

MG-BSP is less applicable for (i) irregular execution such as graph processing, where a task model [81] or an asynchronous model [10] may obtain better parallelization efficiency; (ii) sufficient regular on-chip data reuse exist across supersteps, where tiling may offer better performance than MG-BSP.

III. Density Matrix Quantum Circuit Simulation

In this section, we show how the MG-BSP model can be leveraged for density matrix quantum circuit simulation, bringing desired performance, programmability and applicability. We discuss the general applicability of MG-BSP in Section V.

A. Problem Definition and Algorithm Design

Recall that the density matrix simulation is to compute ρ_{out} for a n-qubit quantum register, given initial state ρ_{in} and m gate transformations $U_0, U_1, \ldots, U_{m-1}$:

$$\rho_{\text{out}} = U_{m-1} \cdots (U_1 (U_0 \rho_{\text{in}} U_0^\dagger) U_1^\dagger) \cdots U_{m-1}^\dagger \tag{2}$$

where U and ρ are $2^n \times 2^n$ matrices. Each U is a unitary matrix verifying $UU^\dagger = U^\dagger U = I$. I is the identity matrix. $U^\dagger$ is the adjoint of U with $U^\dagger = (U^*)^T$.

Given p GPUs, if we directly implement Eq 2, the computation workload for a single gate ($\rho = U\rho U^\dagger$) is two complex matrix-multiplication (size=$2^n \times 2^n$), which is $2MM \times (2^n)^3$ *complex_mul/MM*$\times 6$ *dp_ops/complex_mul* $= 12 \times 8^n$ double-precision (DPs) operations in sum up, corresponding to $12m \times 8^n/p$ per GPU with m gates. Meanwhile, the memory access number is *(3 matrix_read + 3 matrix_write)*$\times 8^n$*complex*$\times 16$ *bytes/complex* = (96×8^n) bytes for matrix multiplication without data reuse, corresponding to $96m \times 8^n/p$ bytes per GPU. Since each gate computation includes an adjoint operation that transposes a $2^n \times 2^n$ matrix, given m gates, the total communication volume is 4^n *complex*$\times 16$ *bytes/complex*$=4^{n+2}m$ bytes, which is tremendous even under a small n and m. The transposition also brings *(1 matrix_read + 1 matrix_write)*$\times 4^n$ *complex* $\times 16$ *bytes/complex*$=(32 \times 4^n)$ bytes of memory access. For efficiency, we propose the following transformations on Eq 2, which has not been reported by any existing works:

$$\begin{aligned}
\rho_{\text{out}} &= U_{m-1} \cdots (U_1 (U_0 \rho_{\text{in}} U_0^\dagger) U_1^\dagger) \cdots U_{m-1}^\dagger \\
&= U_{m-1} \cdots U_1 U_0 \rho_{\text{in}} U_0^\dagger U_1^\dagger \cdots U_{m-1}^\dagger \\
&= \left((\rho_{\text{in}} U_0^\dagger U_1^\dagger \cdots U_{m-1}^\dagger)^\dagger (U_{m-1} \cdots U_1 U_0)^\dagger \right)^\dagger \\
&= \left((U_{m-1} \cdots U_1 U_0 \rho_{\text{in}}^\dagger)(U_{m-1} \cdots U_1 U_0)^\dagger \right)^\dagger
\end{aligned}$$

So we obtain:

$$\rho_{\text{out}} = (U_{m-1} \cdots U_1 U_0)(U_{m-1} \cdots U_1 U_0 \rho_{\text{in}}^\dagger)^\dagger \tag{3}$$

This is equivalent as we partition every gate operator into two steps, rather than performing Step-1 and 2 for each gate consecutively, we perform Step-1 for all gates at once, transpose and conjugate, and then perform Step-2 for all gates again on the adjoint. Since an error $\mathscr{E}$ may occur when a particular gate U is applied, we show that such a transformation can conserve the semantics even when noise has been introduced. This is equivalent to showing that $\mathscr{E}$ applied on $G(\rho)$ is equivalent to applying $\mathscr{E}$ on the gate operator itself and then applying the erroneous gate to ρ.

Theorem 1. *Let $\mathscr{E}(\rho) = (\sum_i K_i)(\rho)$ be a superoperator representing a quantum noise channel where K_i is a particular*

Kraus operator, let G be an operator representing a unitary gate, then $\mathscr{E}(G(\rho)) = (\mathscr{E}(G))(\rho)$.

Proof.

$$\mathscr{E}(G(\rho)) = (\sum_i K_i)(G(\rho)) \quad \textit{//definition of the superoperator}$$
$$= \sum_i K_i G(\rho) K_i^\dagger \quad \textit{//application of the Kraus operators}$$
$$= \sum_i K_i G \rho G^\dagger K_i^\dagger \quad \textit{//application of the gate}$$
$$= \sum_i (K_i G)\rho(K_i G)^\dagger \quad \textit{//product property of the adjoint operator}$$
$$= (\sum_i (K_i G))(\rho) \quad \textit{//application of the Kraus operators}$$
$$= ((\sum_i K_i)(G))(\rho) \quad \textit{//independence of the gate w.r.t. the index}$$
$$= (\mathscr{E}(G))(\rho) \quad \textit{//definition of the superoperator}$$

□

The transformation in Eq 3 brings two fundamental benefits:

- **Memory Access**: Since U_i is applying to one (or two) qubit(s) while leaving other qubits unchanged, this is equivalent to applying U_i to the target qubit while applying identity operation `ID` to all the remaining qubits. If U_i is applied to the i-th qubit, $U = I \otimes I \otimes^{n-i-1 \text{ times}} \cdots U_i \otimes I \otimes^{i \text{times}} \cdots \otimes I$, where "$\otimes$" is the Kronecker product or tensor product. This implies that for the sub-operation $\rho = U\rho$, if ρ is organized in column-major as shown in Figure 3-(A), we can generate the 2×2 matrix of U_i for 1-qubit gate, or 4×4 matrix of $U_{i,j}$ for 2-qubit gate directly on the GPU side (as the operations for other qubits are `ID` operations) without transferring a $2^n \times 2^n$ matrix U from the host. More importantly, we can avoid the memory access of one input matrix, accounting for $1/3$ of the total memory access in the matrix-product operations. Additionally, by dramatically reducing the number of adjoint operations, we can avoid most of the memory access for the adjoint operations.
- **Communication**: First, the number of adjoint operations in Eq 3 is reduced from m to 2. If we directly initialize GPU memory with $\rho_{in}^\dagger$ rather than ρ_{in}, only a single adjoint remains. As such, we reduce communication times by a factor of m. This brings tremendous performance gain, given transposition or all-to-all communication is very expensive, particularly among multi-nodes, as will be seen later. Second, if we partition ρ along columns (Figure 3-(A)), all the computation can be executed locally in a GPU during the operation $(U\rho)$ — there is no inter-GPU communication required for processing a gate. This benefit gains from the co-design of data structure, parallelization strategy, and the algorithm itself. The row-partition as shown in Figure 3-(B) is analogous to the proposed equation transformation, but requires $U^\dagger$ for each gate, leading to extra complexity. Tile-based partition as shown in Figure 3-(C) demands expensive 2D communication and is out of our choice here.

B. Mapping to MP-BSP Machine

The density matrix quantum simulation matches the MG-BSP programming methodology very well (Section II-C): it performs regular computation in supersteps, with strict and unavoidable dependency among the supersteps. Each superstep represents a gate operation: $\rho = (U \rightarrow \rho)$.

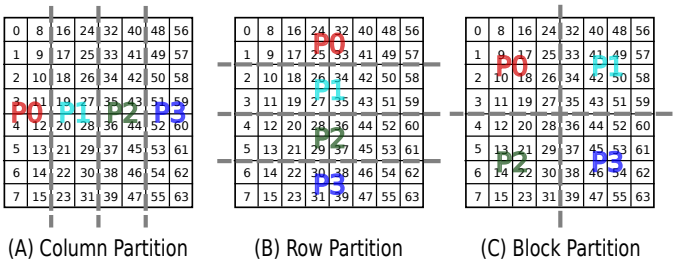

Fig. 3: Partition density matrix ρ among 4 GPUs.

TABLE I: Gates natively supported by the simulator.

Gates	Meaning	Gates	Meaning
`U3`	3 parameter 2 pulse 1-qubit gate	`TDG`	conjugate of sqrt(S) gate
`U2`	2 parameter 1 pulse 1-qubit gate	`RX`	X-axis rotation gate
`U1`	1 parameter 0 pulse 1-qubit gate	`RY`	Y-axis rotation gate
`CX`	Controlled-NOT gate	`RZ`	Z-axis rotation gate
`ID`	Idle gate or identity gate	`CZ`	Controlled phase gate
`X`	Pauli-X bit flip gate	`CY`	Controlled Y gate
`Y`	Pauli-Y bit and phase flip gate	`CH`	Controlled H gate
`Z`	Pauli-Z phase flip gate	`CCX`	Toffoli gate
`H`	Hadamard gate	`CRZ`	Controlled RZ rotation gate
`S`	sqrt(Z) phase gate	`CU1`	Controlled phase rotation gate
`SDG`	conjugate of sqrt(Z) gate	`CU3`	Controlled U3 gate
`T`	sqrt(S) phase gate	`U`	Arbitrary unitary gate

We first describe the ISA Ω, which is the collection of all supported quantum gates. To be general, this simulator is designed to support **OpenQASM** – the *Open Quantum Assembly language* [14]. Consequently, our simulator can run the quantum assembly code compiled from high-level quantum programming languages like Qiskit [30], Cirq [21] and Scaffold [32]. The gates we currently support are the ones defined in the OpenQASM standard header file "*qelib1.inc*" plus an arbitrary unitary gate U, as listed in Table I.

Algorithm 1 shows the computation of applying a quantum gate to a density matrix. The c loop in Line 1, which processes each column of the partition distributed to the current GPU. The outer loop and inner loop iterate along the elements of a column. The outer loop increments in a step depending on the index of the target-qubit for the current operating gate. The inner loop depends on the outer loop, while its iteration range is also dictated by the target-qubit q. Nevertheless, the inner loop iteration step is continuous. The computation pattern in Line 6-7 is *streaming processing*: fetching two complex elements from the density matrix ρ, processing, and writing back to the same location of the density matrix.

With Algorithm 1, we have three observations: (i) The memory accesses depend on per-gate target-qubit index. Therefore, unless performing very sophisticated and time-consuming dependency analysis, it is very difficult to fuse or pipeline across gates; (ii) The c loop, outer loop and inner loop are

Algorithm 1 Simulation workload per GPU per gate

Require: gpu_id, n_gpus, q, n, U_i;
Ensure: $\rho = U_i \times \rho$, $U_i \in \{U_0, U_1, \cdots, U_{m-1}\}$;
1: **for** $c = gpu\_id * \frac{2^n}{n\_gpus}$, ..., $(gpu\_id+1) * \frac{2^n}{n\_gpus} - 1$ **do**
2: **for** $outer = 0$; $outer < 2^n$; $outer += 2 \times 2^q$ **do**
3: **for** $inner = outer$; $inner < outer + 2^q$; $inner{+}{+}$ **do**
4: $V_0 = \rho[c][inner]$;
5: $V_1 = \rho[c][inner + 2^q]$;
6: $\rho[c][inner] = U(V_0, V_1)$; //applying gate U
7: $\rho[c][inner + 2^q] = U(V_0, V_1)$; //applying gate U
8: **return** ρ

```
//Gate ISA format header
#define OP_HEAD grid_group grid = this_grid(); for(...){ ...
//Gate ISA format footer
#define OP_TAIL  } grid.sync();
//Define a Hadamard gate MG-BSP instruction, S2I=1/sqrt(2)
__device__ __inline__ void H_GATE(double* dm_r, double* dm_i,
  const unsigned qubit){
OP_HEAD;
double el0_r=dm_real[pos0]; double el0_i=dm_imag[pos0];
double el1_r=dm_real[pos1]; double el1_i=dm_imag[pos1];
dm_r[pos0]=S2I*(el0_r+el1_r); dm_i[pos0]=S2I*(el0_i+el1_i);
dm_r[pos1]=S2I*(el0_r-el1_r); dm_i[pos1]=S2I*(el0_i-el1_i);
OP_TAIL; }
```

Listing 3: Applying the 1-qubit Hadamard gate.

all independent and parallelizable. Additionally, there is no reuse for the access to the density matrix in the loop-nest; each element of c is updated at most once per gate, demonstrating streaming processing. As will be seen, this kernel is memory-bandwidth bound. Thus, deriving the maximum throughput from the device memory is critical to performance. This is achieved here through *memory coalescing* from the way we formulate the loop nest, and sufficient *memory-level-parallelism* [41], [48] from high occupancy; (iii) The outer and inner loops are not fixed but essentially depend on qubit index q. If we choose to keep them in the GPU kernel function, there may not be sufficient workload to saturate all the thread-slots, particularly when n is small. In fact, for Volta GPU, we need at least 80SM×2048Threads/SM=163,840 threads to saturate all thread-slots. Alternatively, if we choose to map them onto particular dimensions of thread-grid or thread-block, it may hinder kernel fusion. Therefore, we spread out the three loops through index transformation, and generate the thread-block job-lists for the elastic thread blocks to process.

An example about the *Hadamard* gate is shown in Listing 3. Since all the gates follow the same operating process, the simulator defines the *ISA-format-header* and *footer*, which are well-optimized. Therefore, to add a new gate, a domain developer can focus just on the logic of the quantum gate (e.g., Line 9-12 in Listing 3) and offload other jobs like optimization to the simulator framework.

We then define the program P. Figure 1 shows an example of a quantum circuit graph. The circuit can be described by a sequence of gate instructions. Listing 4 shows an example of an 8-bit quantum ripple-carry adder circuit comprising two 4-bit adders built using the instructions from the ISA. As can be seen, it is very easy for the users to define their own gate functions (e.g., *majority* and *unmaj* circuit in Listing 4), as well as call them from another gate function. This is because they are naturally mapped to the MG-BSP subprograms, and eventually handled by the GPU compiling mechanism.

Finally, we define the communication superstep for the simulator. The target is to realize the adjoint operation on a $2^n \times 2^n$ matrix: $A^\dagger = (A^*)^T$. Conjugate is easy, but transpose requires *all-to-all* communication[2] among all GPUs. For instance, given two GPUs G0 and G1, G0 holds data vector [a0, a1] while G1 holds [b0, b1], the all-to-all communication conduct

[2]The *all-to-all* communication in this paper refers to the specific collective `all-to-all()` function equivalent to *MPI_Alltoall()*, rather than the general meaning of all nodes talk to all nodes.

```
__device__ __inline__ void majority(double* dm_real,
  double* dm_imag, const unsigned a, const unsigned b,
  const unsigned c) { CX(c, b); CX(c, a); CCX(a, b, c); }
__device__ __inline__ void unmaj(double* dm_real,
  double* dm_imag, const unsigned a, const unsigned b,
  const unsigned c) { CCX(a, b, c); CX(c, a); CX(a, b); }
__device__ __inline__ void add4(double* dm_real,
  double* dm_imag, const unsigned a1, ...) {
    majority(dm_real, dm_imag, cin, b0, a);
    majority(dm_real, dm_imag, a0, b1, a);
    majority(dm_real, dm_imag, a1, b2, a);
    majority(dm_real, dm_imag, a2, b3, a);
    CX(a3, cout);
    unmaj(dm_real, dm_imag, a2, b3, a);
    unmaj(dm_real, dm_imag, a1, b2, a);
    unmaj(dm_real, dm_imag, a0, b1, a);
    unmaj(dm_real, dm_imag, cin, b0, a);}
__device__ __inline__ void circuit(double* dm_real,
  double* dm_imag) {
    X(2);//a=00000001
    X(10);
    X(16);//b=10111111
    add4(dm_real,dm_imag, 2,3,4,5,10,11,12,13,0,1);
    add4(dm_real,dm_imag,6,7,8,9,14,15,16,17,1,0);
}//Output: 11000000 0
```

Listing 4: 8-bit quantum ripple-carry adder circuit in 18 qubits.

in-place data-exchange so that after the communication, G0 holds [a0, b0] while G1 holds [a1, b1]. As currently NCCL does not support this *all-to-all()* primitive, nor any of the currently supported five primitives can be easily combined to achieve the function, we do not adopt NCCL here. Regarding NVSHMEM, we tried the MPI-based early access version but did not observe particular performance gain. We will wait for more detail about this new library until it is officially released.

For the intra-node peer access design, we adopt the separated kernel strategy — execute the gate sequence once in a kernel, abort and initiate P2P on the CPU-side for the adjoint, and then invoke a new kernel to run the gate sequence again. Note that although our formula reformation reduces the communication transactions to 1, the communication is still critical for performance, as will be seen later. To increase the communication efficiency through larger transfer granularity, we adopt a packing design to compact scattered data dedicated for the same destination into a larger continuous block. Figure 4 illustrates this process with 3 qubits among 4 GPUs. We optimized the *packing*, *unpacking*, and *trans* kernels and formalized them as the helper instructions (see Section II-A and Listing 1) of MG-BSP so they can be fused into the unified global function. Listing 5 shows our design for the *all-to-all* function: For intra-node scenario, we initiate N_GPUS streams from each GPU – one stream accounts for a communication channel to a target GPU peer (i.e. *dst* in Line 5), and rotatively form P2P pairs for efficient peer copying. In this way, these asynchronous streams can overlap with each other (Line 6) and thus being able to fully leverage the inter-GPU interconnect such as NVLink and NVSwitch.

C. DM-Simulator Framework and Toolchain

Figure 5 shows our Density-Matrx (DM) simulation framework and toolchain. There are two paths to generate the quantum circuit file for simulation. One is from the quantum programming languages, where the code is converted to OpenQASM assembly (e.g., *circuit.qasm*) through the frameworks like Qiskit [30], Cirq [21], ProjectQ [78], ScaffCC [32]. Then,

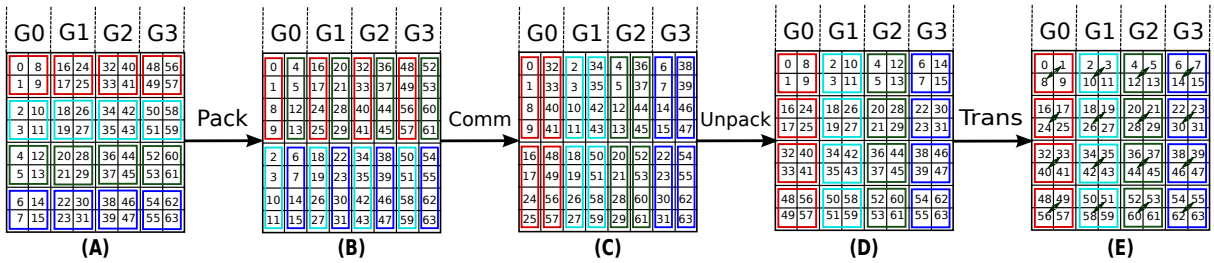

Fig. 4: Inter-GPU communication procedure to achieve density matrix adjoint operation. We adopt two stage transpose: (1) Block transpose via inter-GPU *all-to-all()* communication. (2) local blockwise transpose in a GPU. The blocks in the same color in (A) are transmitted to the same target GPU e.g., red-blocks for G0, blue for G1, green for G2, and pepper for G3. We first pack the discrete segments (e.g., 0, 1, 8, 9) together in (B), perform *all-to-all()* in (C), unpack the block in (D), and conduct blockwise transpose and conjugate gate in (E).

```
1//Intra-node multi-GPU all-to-all comm after packing
2for (int g = 0; g<N_GPUS; g++){
3  unsigned dst = (dev + g) % (N_GPUS);
4  cudaSafeCall(cudaMemcpyAsync(&recv_buf[dst]
5    [M_GPU*M_GPU*dev], &send_buf[dev][dst*M_GPU*M_GPU],
6    COMM_SIZE, cudaMemcpyDefault, streams[dst])); }
7//Intra-node multi-GPU all-to-all comm with MPI
8MPI_Alltoall(send_buf, COMM_SIZE, MPI_DOUBLE,
9  recv_buf, COMM_SIZE, MPI_DOUBLE, MPI_COMM_WORLD);
```

Listing 5: Parallel scale-up and scale-out communication

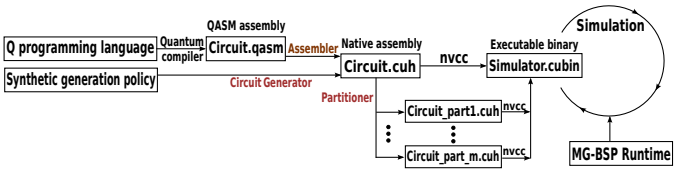

Fig. 5: DM-Sim Simulator Framework.

the assembly is further converted to the native assembly that can be executed by the simulator, through our **Assembler**, which basically (i) translates qubits of independent quantum registers to a unified address space; (ii) recursively maps each user-defined gate to a corresponding subprogram of the MG-BSP machine; (iii) invokes internal gates as listed in Table I. The alternative path is to synthetically generate quantum circuits through our circuit **Generator**, following the user-specified generation policy.

When fusing all gate functions into a single kernel, the CUDA compiler *nvcc* will inline all the device functions into the unified global function. When the number of functions is larger than 2048, the inline process can take extremely long time and crashes occasionally. To simulate very deep circuit (e.g. 1M), we developed a *Partitioner* tool to partition the long gate sequence into multiple chunks, each containing 512 or 1024 gates, and encapsulate each chunk into an independent GPU global function defined in an independent *.cuh* source file. These source files can then be simultaneously compiled by `nvcc` through multi-processing (e.g., via *make -j 32*) and linked into the final executable binary. Figure 5 shows our tool-chain support for the quantum density matrix simulator.

IV. Evaluation

A. Experiment Configuration

We evaluate our DM-Sim quantum circuit simulator using five state-of-the-art multi-GPU platforms: NVIDIA Tesla-P100 DGX-1 (*DGX-1P*) [61], Tesla-V100 DGX-1 (*DGX-1V*), Tesla-V100 DGX-2 [63], RTX-2080 SLI, and ORNL's Summit supercomputer, as listed in Table II. Using these platforms, we cover three GPU generations (*Pascal*, *Volta*, and *Turing*) and five types of interconnect (i.e., *NV-SLI*, *NVLink-V1*, *NVLink-V2*, *NVSwitch* and *InfiniBand*). The SLI-system contains two GPUs. Both DGX-1P and DGX-1V contain 8 GPUs. However, not all of them are directly linked by NVLink. As *all-to-all* communication requires all GPUs being directly connected, we use 4 GPUs (i.e., GPU-0 to 3) in DGX-1P and 1V for the evaluations. We use *CUDA_Event* for the timing measurement. The reported values are average of five times' execution.

Regarding the input quantum circuit (program P), we obtain via two approaches: (i) the circuit generator takes the number of qubit n, the number of gates m, and the set of allowable gate types as inputs. It randomly picks a qubit within $[0, n)$ and randomly chooses a gate from the set. We rely on Python's *unitary_group* function under *scipy.stats* package to generate random unitary gate U; (ii) we rely on the assembler to parse and translate real quantum routines written in OpenQASM to the native assembly supported by our simulator. These sample routines have a fixed number of qubits and gates, which are obtained from the *QASMBench* benchmark suite [38].

B. Roofline-Model Analysis

We use the Roofline model [82] to show the performance bound based on which we can estimate how good performance we have achieved. For theoretic double-precision computation bandwidth (FLOPS), memory bandwidth (GB/s), and interconnect bandwidth (GB/s) per GPU, we use the numbers from the hardware spec-sheet. For the empirical values, we adopt the Empirical-Roofline-Tool (ETR) [57] to collect the sustainable DRAM bandwidth and double-precision FLOPS number. For the interconnect bandwidth, we use the Tartan benchmark suite [43] to collect the maximum suitable bandwidth. It can be a bit surprised to watch that the empirical computation FLOPS collected by ERT for SLI, DGX-1V and DGX-2 are slightly higher than their theoretic FLOPS, possibly due to runtime frequency boosting for advanced GPUs, while the theoretic values are calculated using normal frequency. We draw the roofline model for the kernel of U-gate in Figure 6, which theoretically exhibits the largest arithmetic intensity. As can be seen, the kernel for processing a gate is memory bound across all platforms, implying that in case we approach the memory bound, we may have achieved the best attainable performance.

C. Evaluation Results

We first focus on single-node multi-GPU conditions.

TABLE II: Evaluation Platforms. SP/DP stands for single-precision and double precision. Rtm.stands for runtime library version.

Platform	CPU	Compiler	GPU	GPU Arch	GPUs	Interconnect	Topology	SP/DP GFlops	GPU Memory	Rtm.
SLI	Intel Xeon E5-2680	gcc-4.8.5	RTX-2080	Turing	2	NVLink-SLI	Twin	10068/314.6	8GB GDDR6@448GB/s	10.0
P100-DGX-1	Intel Xeon E5-2698	gcc-4.8.4	Tesla-P100	Pascal	8	NVLink-V1	Hypercube	10609/5304	16GB HBM2@732 GB/s	8.0
V100-DGX-1	Intel Xeon E5-2698	gcc-5.4.0	Tesla-V100	Volta(SXM2)	8	NVLink-V2	Hypercube	14899/7450	16GB HBM2@900 GB/s	9.0
DGX-2	Intel Xeon P-8168	gcc-7.3.0	Tesla-V100	Volta(SXM3)	16	NVSwitch	All-to-All	15551/7776	32GB HBM2@900 GB/s	9.0
Summit	IBM Power-9	xlc-16.1.1	Tesla-V100	Volta(SXM2)	6×4500	NVLink-V2/InfiniBand	Island/Fat-tree	14899/7450	16GB HBM2@900 GB/s	9.2

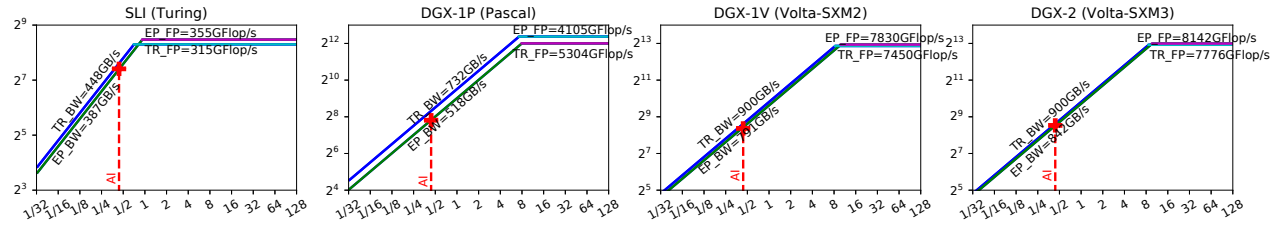

Fig. 6: The Roofline Model. TR_BW stands for theoretic memory bandwidth, EP_BW stands for empirical memory bandwidth, TR_FP stands for theoretic FLOPS, ER_FP stands for empirical FLOPS. "AI" stands for arithmetic intensity of the U gate.

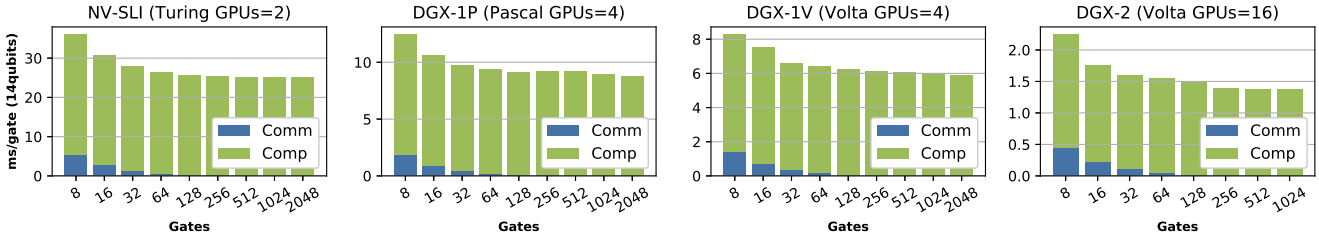

Fig. 7: Average milliseconds per gate with respect to the number of gates simulated ($n = 14$). "Comm" is short for Communication. "Comp" is short for Computation.

TABLE III: Evaluated quantum routines. We use the routines of 17 &18 qubits for scale-out evaluation.

Routine	Description	Qubits	Gates
deutsch	Deutsch algorithm with 2 qubits for $f(x) = x$	5	5
grover	Grover amplification repeated twice	3	123
wstate	W-state assessment	3	30
iswap	Swap the state of qubits	5	9
pea3pi8	Quantum phase estimation algorithm	5	74
qec	Repetition code syndrome measurement	5	5
qv5	Quantum volume analysis	5	100
w3test	Experiment of W-state	5	10
adder	Quantum ripple-carry adder	9/18	139/281
sat	Boolean satisfiability problem	10	679
bv	Bernstein-Vazirani algorithm	15/17	44/50
cc	Counterfeit-coin finding algorithm	15/17	28/30
qft	Quantum Fourier transform algorithm	15/17	540/697

Performance with gate: Figure 7 illustrates the simulation performance (normalized to ms/gate) for randomly generated circuits with respect to the number of gates (m) under 14-qubits over the four single-node platforms. We also show the breakdown of per-GPU computation and inter-GPU communication. As can be seen, the delay per gate reduces with more gates until $m \approx 256$; after that, the delay keeps steady. This is true for both communication and computation. The reason is that the communication delay (blue portion) and the overhead for data packing, unpacking, and block-wise transpose (see Figure 4) are amortized with more gates being simulated. Overall, $m = 256$ is sufficient to achieve the optimal performance. With 14 qubits, the best performance is obtained on DGX-2 with 16 GPUs. The average delay is ~ 1.4 ms/gate.

Performance with qubits: Figure 8 illustrates the delay-per-gate with respect to the number of qubits (from 3 to 15) using 256 gates on the four platforms. As can be seen, the delay keeps steady until certain qubits (i.e., $n = 8$ for SLI, $n = 10$ for DGX-1P and DGX-1V, and $n = 11$ for DGX-2), and then start to increase exponentially (note the Y-axis is in log scale). This is because the problem size scales in 4^n with qubits n. Due to log-based Y-axis, the communication appears to take a much larger portion than expected, but essentially it is very tiny. Compared among platforms, DGX-2 exhibits the largest communication overhead due to more GPUs involved.

Performance bounds: To further evaluate the performance, Figure 9 shows the normalized GPU computation, memory access, and inter-GPU communication bandwidth per GPU for the 4 platforms with respect to the number of qubits. We also draw the empirical bandwidth upper bounds (the dot-line) to see how good we are. Figure 9-(A) is to compare among three GPU architectures: *Pascal*, *Volta*, and *Turing*. As can be seen, DGX-2's Volta-SXM3 shows the best performance, slightly better than DGX-1V's Volta-SXM2. Meanwhile, both Volta and Pascal are much better than Turing. This is because Turing is mainly for desktop utilization, the double-precision performance is much lower ($\sim$314.6 FLOPS). Overall, the delivered computation throughput is below the bound. Figure 9-(B) is to compare with the three types of GPU DRAM: GDDR-6 in RTX-2080, P100-HBM2 in DGX-1P, and V100-HBM2 in DGX-1V and DGX-2. The figure confirms that the execution is memory bound and we have already achieved the optimal performance that can be obtained under this algorithm design. Figure 9-(C) is to compare among the four types of multi-GPU interconnect: NV-SLI in the SLI-system, NVLink-V1 in DGX-1P, and NVLink-V2 in DGX-1V and DGX-2. We almost touch the interconnect bandwidth bound with 15 qubits.

Performance scaling with GPUs: Figure 10 illustrates the

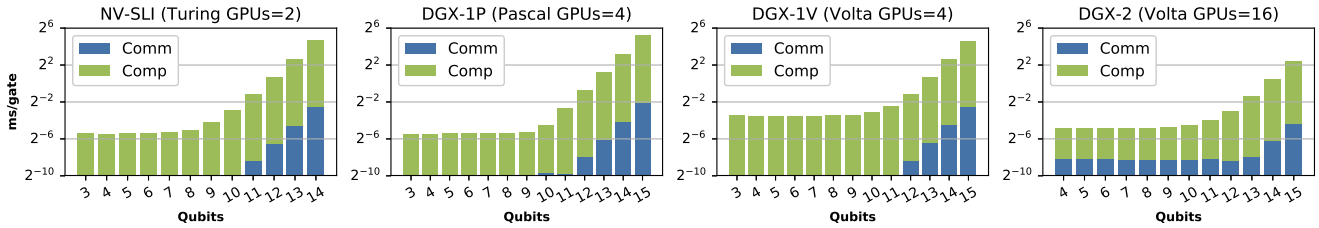

Fig. 8: Average milliseconds per gate with respect to the number of qubits simulated ($m = 256$).

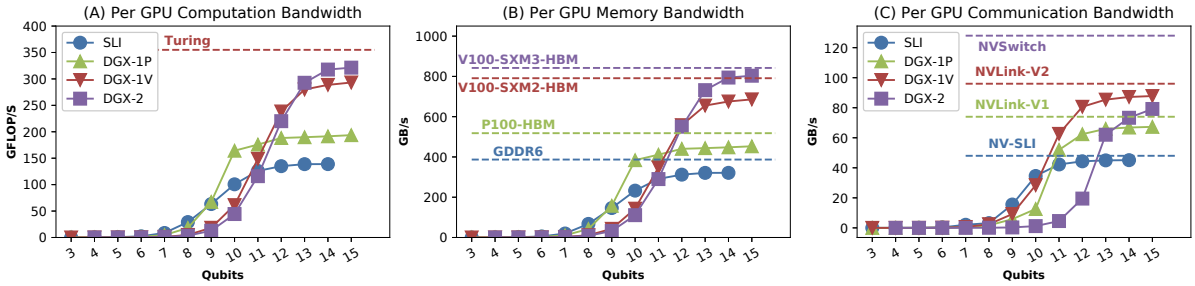

Fig. 9: Computation, memory and communication bandwidth for the DM-Sim simulator with respect to qubits on the four single-node platforms. We have approached the memory bound, and for interconnect, we also approach the bound except DGX-2.

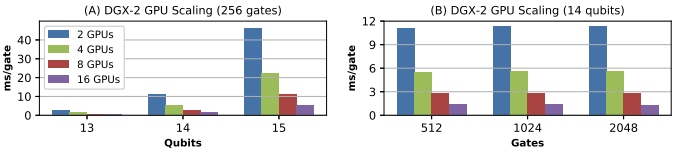

Fig. 10: GPU scaling evaluation for our simulation on DGX-2.

TABLE IV: Deep simulation on DGX-2 using 15 qubits.

Gates	Comp	Comm	Sim	Comp/gate	Comm/gate	Sim/gate
10K	53.8s	9.36ms	53.8s	5.377ms	0.936us	5.378ms
100K	558.0s	7.31ms	558.0s	5.58ms	0.073us	5.580ms
1M	5645.5s	7.21ms	5645.5s	5.65ms	0.007us	5.65ms

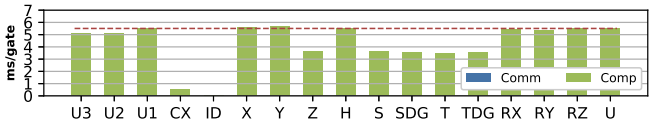

Fig. 11: Average delay per gate for basic simulator built-in gate types on DGX-2 with n=15 qubits and m=256 gates.

average delay per gate with respect to the number of GPUs adopted in the simulation *(GPUs = 2, 4, 8, 16)* on DGX-2. The delay halves each time we double the number of GPUs, demonstrating strong scaling capability of the simulator.

Deep simulation: We perform extremely deep quantum circuit tests on DGX-2. Since it takes extremely long time for nvcc to inline 1-million gate kernels, we rely on the *partitioner* tool (see Figure 5) to segment the circuit file into nearly 2000 subprograms, each per 512 gates, and in an independent header file. We then compile these files in parallel. We increase the number of gates m from 10K to 100K to 1M. The results are listed in Table IV. Our simulator is able to accomplish 1 million density matrix quantum gate simulation within $5,645s$ or 1.5 hours, on average 5.65 $ms/gate$, under 15 qubits. Such a deep simulation has not been reported by existing simulators.

Gate types: For the previous evaluations, we use the circuit

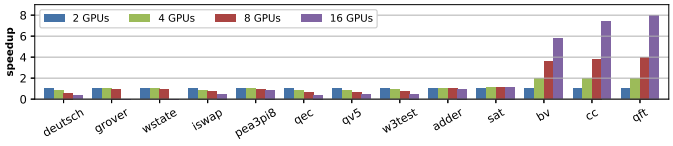

Fig. 12: Strong scaling for quantum routines in Table III on DGX-2.

generator to randomly generate U gates to simulate arbitrary erroneous gate operations. Now we test each built-in gate type specifically. Note that some of the built-in gates (i.e., `CZ`, `CY`, `CH`, `CCX`, `CRZ`, `CU1`, `CU3`) are composed of other basic gates (i.e., the remaining gates in Table I). Figure 11 illustrates the delay per gate for all the basic gates natively defined in the simulator as well as OpenQASM's standard header file. As can be seen, `U1`, `X`, `Y`, `H`, `RX`, `RY`, `U` exhibit similar delay. `ID` gate does not perform any real job, so the delay is nearly zero. `CX` shows much less delay because whether `CX` operation is applied depends on the value of the control qubit, which may not necessarily be 1 given random initialization of ρ.

Algorithm Test: Figure 12 illustrates the speedups of using more GPUs for real quantum routines listed in Table III on DGX-2. As can be seen, for small-scale circuits with 3-5 qubits, using multiple GPUs does not bring performance benefit. On the contrary, for some of the cases (e.g., *deutsch*, *iswap*, *qec*, *qv5*, and *w3test*), the performance even drops with more GPUs. This is because the computation workload is so light that the GPU processors are quite underutilized where latency cannot be effectively hidden. Meanwhile, communication can take a significant portion of the total simulation time. Since the communication delay generally increases with more GPUs involved, we observe such performance degradation. For medium-scale circuits (e.g., *adder*), the GPU utilization is much better, as more gates also help to amortize the communication delay. For large-scale deep circuits, we observe nearly ideal strong scaling speedups, especially for *qft*.

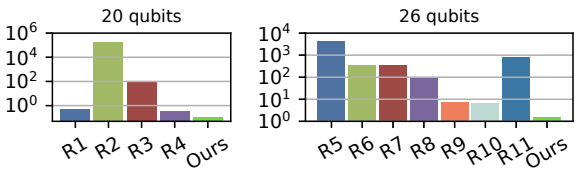

Fig. 13: Performance Comparisons with 10 other QC simulators: R1 [34], R2 [6], R3 [4], R4 [16], R5 [76], R6 [23], R7 [22], R8 [7], R9 [77], R10 [33], R11 [86].

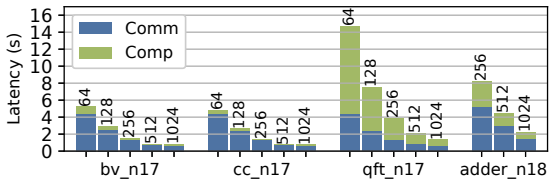

Fig. 14: Simulation delay for *bv*, *cc*, *qft* and *adder* (see Table III). The number on top of the bars indicate the number of GPUs used in the simulation. Note each Summit node comprises 6 Volta GPUs and only 256 GPUs provide enough GPU memory for 18 qubits.

Comparisons: Figure 13 shows the performance (normalized to *ms* per gate) comparisons with other quantum simulators. As the density matrix scales in 4^n, 15 qubits density matrix simulation is equivalent to 30 qubits state vector simulation, as has been shown by existing works [33]. We normalize 10 referencing simulators' performance using the performance merits reported in their papers into two systems (20 and 26 state-vector qubits) for comparison. Our simulator delivers more than 10× performance advantages than its counterparts, demonstrating the effectiveness of our simulator design.

Performance scaling out on Summit HPC: To evaluate our MG-BSP based DM-Sim simulator on multi-node GPU clusters, we test it on ORNL's Summit HPC. We use MPI *all-to-all()* for the communication (see Listing 5) with GPUDirect RDMA enabled (*smpiargs="-gpu"*). We test 17-qubits *bv*, *cc*, *qft* and 18-qubits *adder* (see Table III) using 64, 128, 256, 512, 1024 GPUs on Summit. Figure 14 shows the results. As can be seen, both the computation and communication exhibit strong scaling. For the computation, the performance increases due to increased parallel processing resources. For the communication, the latency also reduces largely because GPUs in Summit are connected in a fat-tree topology, more GPUs implies more local links between GPUs, and therefore expanded overall network bandwidth for migrating the huge density matrix. Consequently, we see strong scaling for communication. Meanwhile, we observe that the percentage of communication is tremendously larger than within a single node. Also note that this is just for a single adjoint operation, without the proposed formula transformation (see Eq 3), the simulation will be completely dominated by communication. For such large-scale simulations, to amortize the expensive communication overhead, deep simulation is always preferred.

V. Discussion

In this section, we discuss the application generality and platform portability of the MG-BSP programming model. In addition, we further discuss GPU-centric designing paradigm.

A. Application Generality

MG-BSP is a general multi-GPU programming methodology that can be applied to many other scenarios in addition to the DM-Sim simulator, particularly those computation systems with indivisible or meta operations from a certain operation set. For example, in a complex sparse numerical solver, the input tensor has to be processed by a series of sparse linear algebra operators such as SpMV [56], SpGEMM [53], SpTrsv [54], [55], etc. Each sparse operator can be defined as an MG-BSP instruction sharing the same parallelization configuration with other instructions. The collection of them form the ISA. The operator sequence in the solver formula becomes the program provided by the users. Another ideal example is the training & inference of deep-neural-networks (DNNs) [37], [47] in high-performance machine learning. Each layer function, such as convolution, fully-connected, pooling, batch-normalization, drop-out, etc. can be defined as an MG-BSP instruction, the user-defined network structure thus is the MG-BSP program. For both examples and other potential scenarios, by following the MG-BSP programming methodology, it can be much easier to manage the synchronization, communication, and the computation logic, following the classical BSP model, while expecting superior performance.

B. Platform Portability

Although this work mainly demonstrates the applicability of the MG-BSP methodology for NVIDIA multi-GPU platforms, there is no obstacle presenting us from applying the methodology to other platforms. Regarding a CPU-cluster such as Theta in ALCF [29], migrating the MG-BSP based implementations such as the DM-Sim simulator are uncomplicated. Particularly, we need to replace the CUDA device functions with OpenMP annotated nested loops, and possibly accelerated by CPUs' vector-instructions (e.g. SSE/AVX). Alternatively, for AMD GPUs, as soon as the corresponding global synchronization mechanism similar to `grid.sync()` can be integrated into the OpenCL or the HIP framework, we can quickly transform the CUDA-based implementations, such as the present DM-Sim simulator, to the AMD CPU/GPU based cluster platform, e.g., the forthcoming ORNL Frontier supercomputer, through the HIP tool [5]. The strong support of GPU-initiated communication mechanism [25], [36] can further benefit the efficiency of the communications in MG-BSP.

C. GPU-centric Designing Paradigm

We envision the forthcoming GPU-centric programming paradigm could be realized through the following aspects: (a) Non-interruptive long GPU-centric computation. All the functions of an application should be realized as device functions and fused into as few kernels as possible to fully leverage the GPU computation resources. Consequently, we can avoid a lot of data migration, communication, synchronization, repeated kernel invocation & release overhead; (b) GPU-centric communication and synchronization. This is a co-design approach: on the hardware side, following the current trend, the inter-node network interfaces should be directly connected to the HBM or NoC of the GPUs rather than a channel of the PCI-e bus (which is the current practice of Summit); from the

software, GPU should be able to initiate intra- and inter-node communication transactions without CPU intervention [25], [36]. Furthermore, GPU device-wide and inter-device synchronization primitives, including global barriers and remote atomic operations, should be offered and callable from the GPU-side; (c) GPU-centric memory access. A GPU should be able to easily access the memory of remote GPUs through either a shared-memory model or message-passing model without incurring too much overhead; (d) GPU-centric I/O. Ideally, GPU can fetch from and store data to various storage including system memory, on-package memory [39], extended memory (e.g., Intel Optane memory [31]) and even file-system, from the GPU-side. Within the four aspects, (a) to (c) are reflected in the MG-BSP programming model of this work.

VI. Related Work

There exist many simulators for simulating quantum circuits in a classical computer, as can be found in the zoo [1]. They have different capabilities, features and restrictions. However, most of them are based on state vectors [4], [6], [7], [15], [16], [20], [22]–[24], [27], [28], [34], [75], [77] and are focusing on logic qubits where full gate fidelity is ensured. For them, pursuing more qubits is one of the major targets. While state vector simulations have general-purpose usage, density matrix simulations are required if mixed-state quantum systems are to be analyzed, particularly with noise [66]. This is the major **motivation** we build a new QC simulator. Additionally, many existing simulators only simulate from a single gate to a few hundreds of gates [15], [22], [24], [26], [34], [68]. However, practical quantum algorithms like variational algorithm [69] demand ultra-deep circuits with millions of gates (10^{14} [73], 10^{18} gates [65]). Besides, with such deep circuits, performance can be a major challenge. This also motivates us to build the MG-BSP based DM-Sim simulator.

Classical simulators come in two flavors with respect to the utilization of the underlying hardware: some simulators use only CPUs to perform the task [15], [20], [28], [75], [77], whereas others provide functionality to leverage readily available heterogeneous resources such as GPUs [4], [6], [7], [16], [22]–[24], [27], [33], [34], [76]. However, the majority of these GPU works use a single GPU. At the time of writing the paper, we are aware of three works leveraging multiple GPUs for quantum simulation [16], [52], [86]. Zhang et al. [86] exploited a single node with 4 NVIDIA Kepler K20 GPUs for state-vector quantum simulation. They proposed well-designed locality-focused communication schemes. However, the round-trip communications between the host and each device could not be avoided for every alternating gate pair consisting a gate operating on local qubits and one operating on remote qubits, as they attempted to explore inter-gate data reuse. In addition, their multi-GPU programming model is the conventional CPU-centric map-reduce model, where for each iteration, the data and job have to be distributed from the CPU master to their GPU slaves. The communication is only between CPU and GPUs via the PCI-e interconnect. Li and Yuan [52] observed that the number of GPUs in a node can be limited by PCI-e and proposed to rely on GPU cluster for state-vector quantum simulation, while trying to reduce communication frequency and the amount of communication data by exploiting locality through a novel data distribution method. They validated their approach on a cluster with 4 nodes and each node contained 4 NVIDIA Kepler K20 GPUs. More recently, Doi et al. [16] proposed a state-vector quantum simulator for multi-nodes multi-GPUs. Their major observation is that a CPU-GPU hybrid design can leverage the strong compute capability of GPUs for efficient simulation, while the large system memory of CPUs for accommodating more qubits. Their strategy was to first allocate the state-vector on GPU memory, and in case insufficient, they partitioned the simulation of a gate into chunks, while carefully managing the chunk communication and synchronization among CPUs and GPUs. Their testing platform was analogous to the node of Summit (IBM Power CPU with 6 NVIDIA Volta V100 GPUs). Their testing ran up to 32 nodes. All of the three works focused on state-vector simulation (thus cannot handle mixed-state quantum systems) and attempted to improve computation or communication efficiency by exploiting data locality. Their designs all include repeated GPU kernel invocations with significant CPU involvement (i.e., CPU-centric). Their evaluations are performed on a single node or relatively small/medium-scale GPU clusters. Comparatively, our DM-Sim simulator focuses on density-matrix simulation and follows the GPU-centric MG-BSP model with merely one or two GPU kernel calls. We rely on an algorithm/data-structure co-design approach to reduce communication and memory access overhead with little CPU intervention. Our design greatly benefits from the novel inter-GPU interconnect and GPUDirect-RDMA, demonstrating strong scaling for both scaling-up (up to 16 GPUs in DGX-2) and scaling-out (up to 1024 GPUs on Summit) scenarios.

VII. Conclusion

In this paper, we propose a novel multi-GPU programming methodology called MG-BSP, which uniforms the thread configuration and integrates all computation and communication supersteps into a single GPU kernel for full occupancy and superior performance. We then apply MG-BSP to build a density matrix quantum simulator DM-Sim. We propose a new simulation approach that significantly reduces communication overhead. Extensive evaluations on five state-of-the-art multi-GPU platforms demonstrate the efficiency, flexibility and scalability of our density matrix simulator. We believe our simulation approach enables effective validation and study the impact of noise on very deep quantum circuits that are expected as we move toward quantum computations beyond the realm of classical computers.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. This research was supported by PNNL's Quantum Algorithms, Software, and Architectures (QUASAR) LDRD Initiative. This research used resources supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

REFERENCES

[1] List of QC simulators. https://www.quantiki.org/wiki/list-qc-simulators.

[2] Quantum Computing: Qubit and Entanglement. http://universe-review.ca/R13-11-QuantumComputing.htm.

[3] Quantum Computing Report. https://quantumcomputingreport.com/scorecards/qubit-quality/.

[4] Andrei Amariutei and Simona Caraiman. Parallel quantum computer simulation on the GPU. In *15th International Conference on System Theory, Control and Computing*, pages 1–6. IEEE, 2011.

[5] AMD. AMD HIP Toolchain, 2020.

[6] Anderson Avila, Adriano Maron, Renata Reiser, Mauricio Pilla, and Adenauer Yamin. GPU-aware distributed quantum simulation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 860–865. ACM, 2014.

[7] Anderson Avila, Renata HS Reiser, Mauricio L Pilla, and Adenauer C Yamin. Optimizing D-GM quantum computing by exploring parallel and distributed quantum simulations under GPUs arquitecture. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 5146–5153. IEEE, 2016.

[8] Ryan Babbush, Jarrod McClean, Dave Wecker, Alán Aspuru-Guzik, and Nathan Wiebe. Chemical basis of Trotter-Suzuki errors in quantum chemistry simulation. *Physical Review A*, 91(2):022311, 2015.

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

[10] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *ACM SIGPLAN Notices*, volume 52, pages 235–248. ACM, 2017.

[11] Iulia Buluta and Franco Nori. Quantum simulators. *Science*, 326(5949):108–111, 2009.

[12] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. Bulk synchronous parallel computinga paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, 1996.

[13] Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo, and Guo-Ping Guo. 64-qubit quantum circuit simulation. *Science Bulletin*, 63(15):964–971, 2018.

[14] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

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

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

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

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

[19] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[20] Vlad Gheorghiu. Quantum++: A modern C++ quantum computing library. *PloS one*, 13(12):e0208073, 2018.

[21] Google. URL: https://github.com/quantumlib/Cirq.

[22] Eladio Gutierrez, Sergio Romero, Maria A Trenas, and Emilio L Zapata. Simulation of quantum gates on a novel GPU architecture. In *International Conference on Systems Theory and Scientific Computation*, 2007.

[23] Eladio Gutierrez, Sergio Romero, Maria A Trenas, and Emilio L Zapata. Parallel quantum computer simulation on the CUDA architecture. In *International Conference on Computational Science*, pages 700–709. Springer, 2008.

[24] Eladio Gutiérrez, Sergio Romero, María A Trenas, and Emilio L Zapata. Quantum computer simulation using the CUDA programming model. *Computer Physics Communications*, 181(2):283–300, 2010.

[25] Khaled Hamidouche and Michael LeBeane. 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*, pages 336–347, 2020.

[26] Thomas Häner and Damian S Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2017.

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

[28] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018.

[29] Kevin Harms, Ti Leggett, Ben Allen, Susan Coghlan, Mark Fahey, Carissa Holohan, Gordon McPheeters, and Paul Rich. Theta: Rapid installation and acceptance of an xc40 knl system. *Concurrency and Computation: Practice and Experience*, 30(1):e4336, 2018.

[30] IBM. An open-source quantum computing framework for leveraging today's quantum processors in research, education, and business. https://qiskit.org/.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[32] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 1. ACM, 2014.

[33] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. Quest and high performance simulation of quantum computers. *Scientific Reports (Nature Publisher Group)*, 9:1–11, 2019.

[34] A. Billfalk Kelly. Simulating Quantum Computers Using OpenCL. 2018.

[35] Shelby Kimmel, Cedric Yen-Yu Lin, Guang Hao Low, Maris Ozols, and Theodore J Yoder. Hamiltonian simulation with optimal sample complexity. *npj Quantum Information*, 3(1):13, 2017.

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

[37] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. 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*, pages 1–30, 2019.

[38] Ang Li and Sriram Krishnamoorthy. QASMBench: A Low-level QASM Benchmark Suite for NISQ Evaluation and Simulation. *arXiv preprint arXiv:2005.13018*, 2020.

[39] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 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*, pages 1–14, 2017.

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

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

[42] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan Tallent, and Kevin Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and f Systems*, 2019.

[43] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202. IEEE, 2018.

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

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

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

[47] Ang Li and Simon Su. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *arXiv preprint arXiv:2006.16578*, 2020.

[48] Ang Li, YC Tay, Akash Kumar, and Henk Corporaal. Transit: A visual analytical model for multithreaded machines. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 101–106, 2015.

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

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

[51] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014, 2019.

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

[53] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. Register-aware optimizations for parallel sparse matrix–matrix multiplication. *International Journal of Parallel Programming*, 47(3):403–417, 2019.

[54] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer, 2016.

[55] Weifeng Liu, Ang Li, Jonathan D Hogg, Iain S Duff, and Brian Vinter. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience*, 29(21):e4244, 2017.

[56] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, 2015.

[57] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014.

[58] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

[59] Ajit Narayanan and Mark Moore. Quantum-inspired genetic algorithms. In *Proceedings of IEEE international conference on evolutionary computation*, pages 61–66. IEEE, 1996.

[60] NVIDIA. NVIDIA Collective Communications Library (NCCL-V2). https://developer.nvidia.com/nccl.

[61] NVIDIA. NVIDIA DGX-1 System Architecture White Paper, 2017.

[62] NVIDIA. CUDA Best practice guide, 2018, 2018.

[63] NVIDIA. NVIDIA DGX-2H The World's Most Powerful System for The Most Complex AI Challenges, 2018.

[64] NVIDIA. Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2018.

[65] National Academies of Sciences Engineering, Medicine, et al. *Quantum computing: progress and prospects*. National Academies Press, 2019.

[66] TE O'Brien, B Tarasinski, and L DiCarlo. Density-matrix simulation of small surface codes under current and projected experimental noise. *npj Quantum Information*, 3(1):39, 2017.

[67] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418. ACM, 2013.

[68] Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, and Robert Wisnieff. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.

[69] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014.

[70] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.

[71] Sreeram Potluri, Nathan Luehr, and Nikolay Sakharnykh. Simplifying Multi-GPU Communication with NVSHMEM. In *GPU Technology Conference*, 2016.

[72] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[73] Markus Reiher, Nathan Wiebe, Krysta M Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences*, 114(29):7555–7560, 2017.

[74] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating a file system with gpus. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 485–498, 2013.

[75] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. qHiPSTER: the quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.

[76] Alexander Smith and Khashayar Khavari. Quantum Computer Simulation Using CUDA. http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/QFT_report.pdf, 2008.

[77] Damian S Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *arXiv preprint arXiv:1612.08091*, 2016.

[78] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.

[79] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[80] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE, 2008.

[81] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[82] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[83] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.

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

[85] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(2):9, 2012.

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