Enabling Quantum Computer Simulations on AMD GPUs: a HIP Backend for Google's `qsim`

Stefano Markidis
KTH Royal Institute of Technology
Stockholm, Sweden
markidis@kth.se

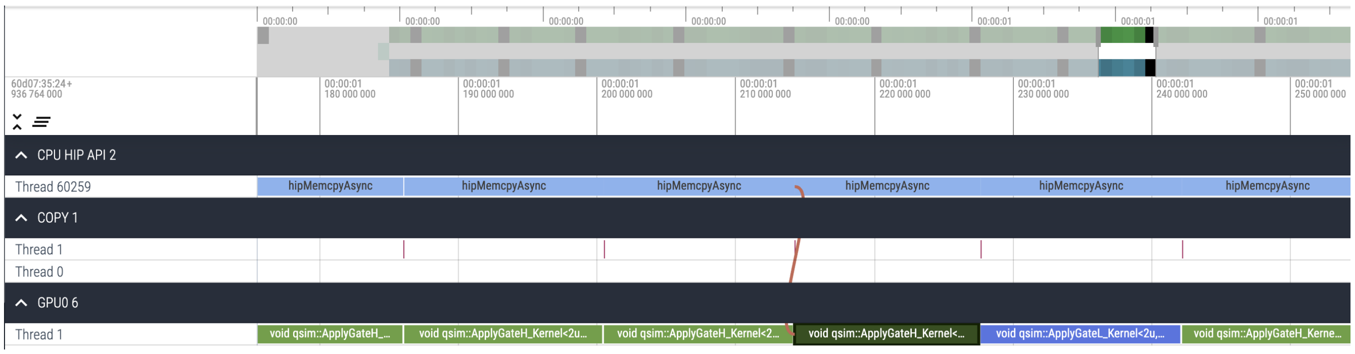


Figure 1: Visualization of the performance traces of `qsim` HIP backend running the Random Quantum Circuit (RQC) sampling benchmark on an AMD MI2050X GPU. The profiling information is obtained with the `rocprof` tool and visualized with `Perfetto UI` on a web browser. The trace visualization shows the execution of two main kernels (`ApplyGateH_Kernel` and `ApplyGateL_Kernel`) and HIP asynchronous memory copies (`hipMemcpyAsync`).

ABSTRACT

Quantum computer simulators play a critical role in supporting the development and validation of quantum algorithms and hardware. This study focuses on porting Google's `qsim`, a quantum computer simulator, to AMD Graphics Processing Units (GPUs). We leverage the existing `qsim` CUDA backend and harness the HIPIFY tool to provide a `qsim` HIP backend tailored for AMD GPUs. Our performance analysis centers on evaluating the HIP backend's capabilities, executed on a computing node equipped with the AMD MI250X GPU and the AMD EPYC Trento CPU. We use the Random Quantum Circuit (RQC) sampling benchmark, employing a circuit featuring 30 qubits. The `qsim` HIP backend on AMD GPU outperforms the CPU version by a remarkable margin, achieving seven to nine times faster speeds. Our investigation also compares `qsim`'s performance on the Nvidia A100 and AMD MI250X GPUs. The Nvidia A100 consistently outperforms the AMD MI250x counterpart, and this performance gap further widens with optimal gate fusion configurations. For instance, a two-gate fusion configuration exhibits a 5% difference, whereas a four-gate fusion setup reveals a large 44% performance gap. Our work highlights the substantial performance advantage of GPU-based quantum simulation over traditional CPU approaches. Despite a performance lag compared to the `qsim` CUDA backend, the AMD HIP `qsim` backend emerges as a competitive alternative poised for further optimization.

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624223

CCS CONCEPTS

• **Computing methodologies → Simulation evaluation**; • **Applied computing → Engineering**.

KEYWORDS

quantum computer simulator, state vector simulator, qsim, AMD GPUs, MI250x, HIP

ACM Reference Format:
Stefano Markidis. 2023. Enabling Quantum Computer Simulations on AMD GPUs: a HIP Backend for Google's `qsim`. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624223

1 INTRODUCTION

Quantum computing has emerged as a disruptive paradigm, offering the potential to tackle problems practically intractable for classical computers. Its applications span various domains, including cryptography with the Shor's algorithm [16], quantum simulations with Quantum Variational Eigensolvers (VQE) [36], and quantum machine learning with Parametrized Quantum Circuits (PQC) [28], among several emerging quantum applications. Along the path toward achieving fault-tolerant error-corrected quantum computers, quantum computer simulators have emerged as indispensable tools.

These simulators play a vital role in designing, validating, and optimizing quantum computing hardware, software, and algorithms. For instance, simulators influence design choices in quantum computer architectures and allow scientists to verify the correctness of quantum algorithms before deploying them on real quantum hardware. Additionally, they enable computer scientists to run quantum programs quickly on workstations without queueing for access to quantum computers during the prototyping phase.

At a high level, two categories of simulators exist: *pulse-level* simulators [25], which simulate quantum hardware at a low-level, modeling quantum transformations via pulses [2], and *gate-based* simulators, which abstract quantum gates as a powerful portability layer [19, 21]. Gate-based simulators are ideal for computer scientists working on high-level algorithm prototyping. Various quantum computer simulation techniques, including state vector [9], density matrix [24], tensor networks [30, 31, 41], and quantum trajectories [20] simulators, offer different capabilities and trade-offs, such as simulations with ideal vs. noisy gates, computational cost, direct access to the quantum state vector information, and available model approximations. This work focuses on a specific simulation technique: the state vector quantum computer simulator. This simulator represents the quantum state of the qubit system as a complex vector. It is critical for quantum algorithm development, allowing continuous inspection and monitoring of the quantum state vector during circuit execution. However, it suffers the problem of exponential growth of memory needed to represent the state vector, limiting in practice to 35-36 qubits quantum computer simulation on modern single-node workstations with Terabyte-size memory systems.

As quantum computing progresses rapidly, the demand for tools to debug and evaluate quantum systems grows. Developing High-Performance Computing (HPC) quantum computer simulators capable of harnessing the power of HPC systems has become increasingly important. Several works focused on developing HPC quantum simulators, some of which leverage the Message-Passing Interface (MPI) to distribute data and computation across different supercomputer computing nodes. Examples include IBM Qiskit [37], Intel-QS [1, 17], HybridQ [26] and QuEST [22] frameworks, which provide MPI-based state vector simulators for supercomputing usage. An important HPC technology, Graphics Processing Units (GPUs), offer significant speed-up in quantum computer simulations, often surpassing CPU performance by more than 40 times [14]. For example, Nvidia's cuQuantum [23, 40] is a CUDA-based library for developing quantum computer simulators on Nvidia GPUs. IBM Qiskit, QuEST, HybridQ, and Google's `qsim` [39], the subject of this paper, support execution on Nvidia GPUs via CUDA, Thrust [4], compiler technologies, such as Google's JAX [5], and cuQuantum. However, support for quantum computer simulator development on AMD GPUs is notably lacking, even though many of the world's largest supercomputers, such as Frontier and LUMI, are equipped with multiple AMD GPUs per node.

This challenge of porting `qsim` to utilize AMD GPUs has motivated our work. We aim to bridge this gap by developing a `qsim` backend that harnesses the AMD HIP (Heterogeneous-Compute Interface for Portability) programming interface and tools. Our contribution extends the capabilities of Google's `qsim` to include execution on AMD GPUs, and we provide a comprehensive performance evaluation. In this article, we present the development of the `qsim` backend for AMD GPUs using the HIP programming interface. Furthermore, we showcase and compare its performance against the already supported Nvidia GPU backend. The paper is structured as follows: Section 2 provides background on Google's quantum software stack, state vector simulators, and the AMD GPU programming environment. Section 3 focuses on explaining the porting of `qsim` to AMD GPUs. Section 4 details our experimental setup, and Section 5 presents the results. Finally, in Section 6, we discuss related work, and in Section 7, we provide a summary and discuss the significance of this work.

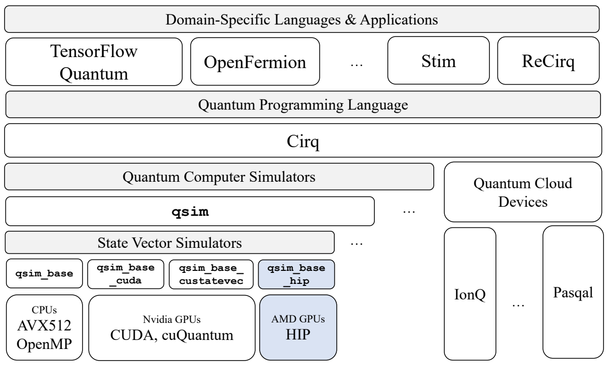


Figure 2: An overview of the full software stack that can use the `qsim` state vector simulator.

2 BACKGROUND

This section provides basic information about the state vector simulator to elucidate the computational task and opportunities for parallelism, presents Google's `qsim` simulator [39], and the AMD tools for programming AMD GPUs we used for porting `qsim` to the AMD GPU systems.

2.1 Google Quantum Computing Software Stack

Before describing the HIP backend's design and implementation for the `qsim` state vector simulator, we first establish the context by elucidating how `qsim` fits into the broader programming software stack capable of using its simulation capabilities. An overview of this software stack is presented in Figure 2. At the top of the quantum software stack are libraries and Domain-Specific Languages (DSLs) specially crafted to develop quantum applications, which can be executed on quantum hardware or simulators. A prime example is TensorFlow Quantum [6], which allows quantum application developers to express quantum neural networks at a higher abstraction level with neural network primitives, eliminating the need for intricate PQC design. This facilitates the rapid creation of quantum machine learning applications [27, 28]. Other noteworthy examples include OpenFermion, an open-source DSL developed by Google tailored for simulating fermionic systems encompassing quantum chemistry and material science [32]; Stim, a library designed for quantum error correction [15]; and ReCirq, a set of modules designed for executing quantum applications and workflows [38].

The bedrock of this quantum software ecosystem is Cirq, an open-source Python-based quantum programming language introduced by Google in 2018 [10]. Similar to other widely adopted quantum programming frameworks like IBM's Qiskit, Cirq is designed for the development of quantum algorithms and simulations. Cirq provides a software platform for designing quantum algorithms in the form of quantum circuits. Moreover, Cirq facilitates the programming of quantum devices, including Google's Sycamore quantum processor. It also includes a suite of tools for optimizing and decomposing quantum circuits to enhance their performance. Cirq further extends its capabilities by offering access to quantum computers via cloud services (providing access to IonQ and PASQAL quantum systems, among many others) and supporting multiple quantum computer simulators.

Notable among these is `qsim`, the central focus of our work, as illustrated in the light blue boxes in Figure 2. `qsim` supports a number of quantum computer simulators. At its core, the *base* simulator within `qsim` provides state vector quantum computer simulation capabilities, featuring ideal gates tailored for various computer architectures. These architectures include CPUs (with support for AVX512 and OpenMP) as well as Nvidia GPUs powered by CUDA and cuQuantum libraries [20]. `qsim` also incorporates a quantum trajectory simulator optimized for modeling noisy circuits, amenable to execution on distributed memory and cloud computing environments. In essence, `qsim` is a C++ library that can be invoked both through Python via Cirq or directly within a C++ application (this is the approach we follow in this work). Accessing the C++ library interface directly from a C++ application provides several advantages, including sidestepping potential cross-compilation challenges that can arise when working with Python. Moreover, this approach allows for comprehensive profiling and tracing of the entire C++ codebase, a task that becomes intricate when dealing with a mixed Python and C++ environment. In addition, the `qsim` can be used as a stand-alone simulator, not requiring the need for the upper part of the software stack presented previously. For these reasons, we opted to employ `qsim` for our porting efforts. In our study, we extend the state vector simulator in `qsim` to use the HIP interface, enabling its utilization with AMD GPUs. This contribution is visually represented in the light blue boxes of Figure 2.

2.2 State Vector Quantum Computer Simulation Techniques

To understand the potential of state vector computer simulators for parallelism and their suitability to accelerators, such as AMD and Nvidia GPUs, it is essential to grasp the fundamental operation of these simulators. Indeed, the formulation of the state vector simulator is straightforward to implement.

We begin by representing the quantum state of our quantum computing system, consisting of n qubits, as a complex-valued column vector of length 2^n, which we refer to as the *state vector*: $[c_1\ c_2\ c_3 \ldots c_{2^n-1}\ c_{2^n}]^T$. Each squared amplitude magnitude of the complex-valued entry i in the state vector corresponds to the probability of measuring the state $|i\rangle$; for instance, $|c_1|^2$ represents the probability of observing the state $|1\rangle$ upon a measurement.

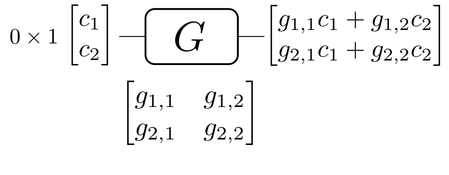

$0 \times 1 \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$
G
$\begin{bmatrix} g_{1,1}c_1 + g_{1,2}c_2 \\ g_{2,1}c_1 + g_{2,2}c_2 \end{bmatrix}$
$\begin{bmatrix} g_{1,1} & g_{1,2} \\ g_{2,1} & g_{2,2} \end{bmatrix}$

Figure 3: In a state vector simulator, the quantum state is calculated by multiplying the unitary matrix representing the quantum gate to the input state vector.

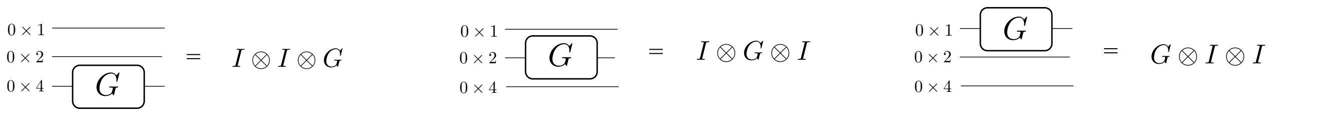
Since these probabilities must sum to one, the state vector must be normalized, as expressed by $\sum_{i=1}^{2^n} |c_i|^2 = 1$.

Quantum algorithms are defined by a sequence of quantum gates acting on the qubit system. In the context of a state vector simulator, these gates are represented as matrices that operate on the state vector. These matrices must be unitary to satisfy the principles of reversibility (involving invertibility) and to preserve the geometry of the system (maintaining $\sum_{i=1}^{2^n} |c_i|^2 = 1$) as dictated by the postulates of quantum mechanics. Similar unitary transformations can be found outside quantum computing, for instance, in computer graphics. For an in-depth understanding of quantum gates and their associated unitary matrices, we recommend referring to quantum computing textbooks [34].

The fundamental quantum computing unit is the *qubit*, a two-state quantum system that can be expressed through a linear superposition of its two orthogonal basis states. A qubit is represented as a two-element complex-value array in a state vector simulator: $[c_1\ c_2]^T$. A quantum gate, denoted as G, is represented by a 2×2 unitary matrix that operates on the input vector $[c_1\ c_2]^T$. The resulting qubit state is obtained by multiplying the matrix G with the input vector $[c_1\ c_2]^T$, as depicted in Figure 3.

By simply inspecting this essential operation at the base of any state vector quantum computer simulator, it is clear that small-size matrix-vector multiplications dominate the state vector simulation calculations. As noted in other papers on HPC state vector simulators [14, 18, 20], the basic computational building block of the state vector simulator is a 2×2 matrix-vector multiplication with only a relatively low arithmetic intensity: a matrix-vector multiplication requires 14 FLOPs and 16 B need to be moved when using single precision.

In practice, all the relevant quantum computing algorithms consist of multiple qubits and quantum gates acting on one or more qubits. In this general case, the state vector is still expressed by a matrix-vector multiplication, albeit the size of the state vector is 2^n. In quantum computing, qubits and quantum gates are composed via tensor product: matrix associated with quantum gates can calculated with the tensor product of the two basic 2×2 quantum gate matrices (or larger sizes in the case of multi-qubit quantum gates). For instance, the basic computational unit is one single qubit gate applied to one qubit in a multi-qubit system: a large part of the circuit calculations can be decomposed into a succession of these basic calculations. The quantum gate acting on a qubit in a multi-qubit system is calculated as the tensor product of the gate


0 × 1
0 × 2
0 × 4
G
= $I \otimes I \otimes G$
0 × 1
0 × 2
0 × 4
G
= $I \otimes G \otimes I$
0 × 1
0 × 2
0 × 4
G
= $G \otimes I \otimes I$

$$\begin{bmatrix} g_{1,1} & g_{1,2} &&&&&& \\ g_{2,1} & g_{2,2} &&&&&& \\ && g_{1,1} & g_{1,2} &&&& \\ && g_{2,1} & g_{2,2} &&&& \\ &&&& g_{1,1} & g_{1,2} && \\ &&&& g_{2,1} & g_{2,2} && \\ &&&&&& g_{1,1} & g_{1,2} \\ &&&&&& g_{2,1} & g_{2,2} \end{bmatrix} \quad \begin{bmatrix} g_{1,1} && g_{1,2} &&&&& \\ & g_{1,1} && g_{1,2} &&&& \\ g_{2,1} && g_{2,2} &&&&& \\ & g_{2,1} && g_{2,2} &&&& \\ &&&& g_{1,1} && g_{1,2} & \\ &&&&& g_{1,1} && g_{1,2} \\ &&&& g_{2,1} && g_{2,2} & \\ &&&&& g_{2,1} && g_{2,2} \end{bmatrix} \quad \begin{bmatrix} g_{1,1} &&&& g_{1,2} &&& \\ & g_{1,1} &&&& g_{1,2} && \\ && g_{1,1} &&&& g_{1,2} & \\ &&& g_{1,1} &&&& g_{1,2} \\ g_{2,1} &&&& g_{2,2} &&& \\ & g_{2,1} &&&& g_{2,2} && \\ && g_{2,1} &&&& g_{2,2} & \\ &&& g_{2,1} &&&& g_{2,2} \end{bmatrix}$$

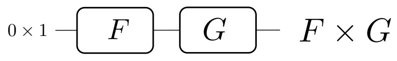
Figure 4: Example of matrices associated with a quantum gate applied to one of the qubits in a three-qubit system. We omit the zero matrix entries and show that the matrices are largely sparse with block and diagonal structures. The structure of these quantum gates' transformation allows for in-place matrix formulation and parallel execution, e.g., each of the four threads can calculate a 2×2 matrix multiplication in parallel, albeit data shuffling might be needed.

matrix with identity matrices. Figure 4 shows an example of resulting matrices corresponding to applying single-qubit quantum gates on different qubits in a three-qubit system: on the leftmost panel, the gate G is applied to the *highest* qubit (0 × 4 in hexadecimal), while on the rightmost panel, the gate G is applied the *lowest* qubit (0 × 1). In this three-qubit system, the state vector is represented by a complex vector of size eight, and the matrix associated with the gate transformation is an 8×8 matrix. The resulting state vector is obtained by multiplying these matrices by the input column vector. Multi-qubit gates, such as controlled versions of single-qubit gates, and their compositions obey the same rules.

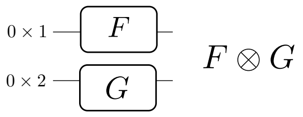
By inspecting Figure 4 showing quantum gate matrices for a 3-qubit system (this is easily generalizable to any n-qubit system) where we omit the zeros, we can deduce three important aspects:

(1) The matrix associated with quantum gate transformations acting on the state vector is predominantly sparse, containing only the quantum gate matrix elements. Consequently, state vector simulators do not store the entire quantum gate matrix, which would also be impractical from a memory perspective. Instead, a matrix-free approach is employed, and the operation is performed in place.
(2) Since the fundamental calculation essentially involves a large matrix-vector multiplication, it can be parallelized by simultaneously computing two elements of the output state vector through 2×2 matrix multiplications. Alternatively, even finer granularity tasks are possible, with each thread dedicated to calculating a single element of the output state vector. As a result, the calculation exhibits a substantial potential for parallelism, either 2^n (one output element per thread) or 2^{n-1} (one matrix multiply per thread).
(3) When a quantum gate is applied to a *lower* quantum bit (for instance, 0 × 1 and 0 × 2 in Figure 4), the calculation necessitates an irregular data access pattern with a stride that depends on which qubit the quantum gate is applied to. In such cases, temporary arrays are employed to store data that is accessed contiguously but with a specific stride. In these scenarios, significant performance enhancements are achieved by utilizing shared memory in Nvidia and AMD GPUs, substantially reducing memory traffic to the global memory.

To further increase the performance of state vector quantum computer simulators, a method known as *gate fusion* can be employed to combine quantum gate matrices into larger matrices. Fusion can occur either in terms of space or time. As previously discussed, two quantum gates that operate concurrently on different qubits can be combined through tensor products. Additionally, when two quantum gates act on the same qubit, they can be fused by directly multiplying their corresponding matrices (while adhering to the order, as matrix multiplication is not commutative). These two kinds of gate fusion for two single-qubit gates are represented in Figure 5.



$$\begin{bmatrix} f_{1,1}g_{1,1} + f_{1,2}g_{2,1} & f_{1,1}g_{1,2} + f_{1,2}g_{2,2} \\ f_{2,1}g_{1,1} + f_{2,2}g_{2,1} & f_{2,1}g_{1,2} + f_{2,2}g_{2,2} \end{bmatrix}$$



$$\begin{bmatrix} f_{1,1}g_{1,1} & f_{1,1}g_{1,2} & f_{1,2}g_{1,1} & f_{1,2}g_{1,2} \\ f_{1,1}g_{2,1} & f_{1,1}g_{2,2} & f_{1,2}g_{2,1} & f_{1,2}g_{2,2} \\ f_{2,1}g_{1,1} & f_{2,1}g_{1,2} & f_{2,2}g_{1,1} & f_{2,2}g_{1,2} \\ f_{2,1}g_{2,1} & f_{2,1}g_{2,2} & f_{2,2}g_{2,1} & f_{2,2}g_{2,2} \end{bmatrix}$$

.

Figure 5: Fusion of two single-qubit quantum gates acting on the same qubit (top panel) and two gates operating in parallel (bottom panel).

These fusion techniques can intensify the computational load of matrix multiplications, albeit at the expense of reducing available parallelism. The process of gate fusion is often considered one of the most crucial optimizations, especially for achieving peak performance on GPUs. Typically, this fusion is carried out by a quantum transpiler, which thoroughly analyzes the quantum circuit to identify and execute gate fusion operations and perform fusion given as input the maximum of fused gates.

2.3 `qsim` Nvidia GPU Backend

`qsim` leverages a highly optimized CUDA codebase for the execution of quantum circuit simulations utilizing the state vector methodology. The foundation of `qsim`'s CUDA backend can be traced back to its AVX512 implementation for CPU vector instructions. Within this backend, the real and imaginary components of 32 state vector amplitudes are maintained in separate sections of the GPU's shared memory, localized to the GPU streaming multiprocessor. This strategic design allows 32 CUDA warp threads, the fundamental scheduling units, to operate on these components concurrently.

The `qsim` GPU backend divides the qubit indices into two categories: those greater than or equal to $\log_2(32) = 5$ are categorized as *high* qubit indices, while those smaller than $\log_2(32) = 5$ are classified as *low* qubit indices. This division is motivated by the relatively straightforward matrix multiplication required for higher-index qubits, in contrast to the more intricate operations needed for lower-index qubits [20]. These operations often necessitate data layout rearrangement and additional memory storage.

Two key kernels are critical in the `qsim` GPU implementation: `ApplyGateH_Kernel` and `ApplyGateL_Kernel`. The former is optimized for quantum gates primarily involving qubits with high indices (greater than or equal to 5), while the latter is tailored for quantum gates involving qubits with low indices. To optimize memory usage and reduce memory traffic, `qsim` stores 32 vector amplitudes and necessary auxiliary variables in the GPU shared memory.

Furthermore, `qsim` employs an additional performance enhancement technique: asynchronous memory copies. This approach enables the overlap of computation and data movement, leading to more efficient processing and resource utilization.

2.4 Programming AMD GPUs

This work is centered on porting the `qsim` quantum computer simulator from its current Nvidia GPU architecture to AMD GPU architecture. In AMD GPU programming, developers can use the AMD Radeon Open Compute (ROCm) framework, which constitutes a comprehensive software ecosystem tailored for AMD GPUs. The ROCm platform encompasses various programming paradigms, including a heterogeneous interface for cross-platform compatibility known as HIP (Heterogeneous-Computing Interface for Portability), GPU offloading using OpenMP directives, and the SYCL programming model. AMD initiated the open-source project HIP in 2016 with the goal of simplifying GPU code development, enabling compilation for both CUDA and ROCm platforms. In this work, we leverage AMD HIP, which provides a C++ runtime API and a kernel language closely aligned with Nvidia CUDA's syntax. HIP essentially substitutes the `cuda` prefix in CUDA functions with `hip`. This transition is motivated by the AMD HIP programming framework's primary objective of streamlining the migration of CUDA-based applications to AMD GPUs. The HIPIFY tools are available to automate the source code conversion from CUDA to HIP for adapting pre-existing CUDA codebases.

Furthermore, HIP offers an assortment of libraries and tools tailored for GPU programming, including ROCm-optimized libraries, such as BLAS, FFT, RNG, Sparse, NCCL (RCCL), and Eigen, among others.

3 IMPLEMENTATION

The design and implementation of the `qsim` backend for AMD GPUs begin with the existing C++ and CUDA state vector backend (called `qsim_base_cuda`) as our starting point. We leverage the HIP programming interface to achieve seamless integration with AMD GPUs. Our first step in the implementation process involved converting the existing CUDA files to HIP files using `hipify-perl`, a tool for porting CUDA code to be compatible with multiple GPU architectures, including AMD GPUs. This automated transformation ensures that the codebase retains its functionality while being compatible with the HIP interface. The conversion process is straightforward, with the `hipify-perl` tool handling most of the necessary adjustments, significantly reducing the manual effort required. There are seven `qsim` library files affected by the conversion:

(1) `qsim_base_cuda.cu` → `qsim_base_hip.cpp`: This code is a C++ program for simulating quantum circuits, taking the circuit configuration as an input file. It uses the `qsim` GPU backend and allows users to specify various simulation parameters via command-line options, e.g. maximum number of fused gates.
(2) `simulator_cuda.h` → `simulator_hip.h`: This code includes a C++ class containing methods for the quantum circuit simulation, such as `ApplyGate`, and `ApplyControlledGate`. These functions perform quantum operations on the quantum state and launch the GPU kernels.
(3) `simulator_cuda_kernels.h` → `simulator_cuda_kernels.h`: These C++ files include the actual implementation of the CUDA/HIP kernels, such as `ApplyGateH_Kernel` and `ApplyGateL_Kernel`, and their controlled gate versions. Regarding GPU performance optimization, we note the usage of shared memory to decrease the memory traffic to the global memory and warp-level reduction operations [7].
(4) `state_space_cuda.h` → `state_space_hip.h`: The class defines various methods for quantum state manipulation and operations, including functions for setting states, getting amplitudes, performing mathematical operations on states, measuring quantum states, inner product, reduction, vector addition, and multiplication. These methods are responsible for launching kernels.
(5) `state_space_cuda_kernels.h` → `state_space_hip_kernels.h`: These files include GPU kernel functions for performing reduction operations on arrays of complex numbers, setting the values of elements in the quantum state array, performing element-wise addition and multiplication, and sampling from a quantum state to generate measurement outcomes.
(6) `cuda_util.h` → `hip_util.h`: These files provide utility functions and data structures for handling CUDA errors, working with complex numbers, and performing warp/wavefront-level reductions in CUDA/HIP kernels.

(7) `vectorspace_cuda.h` → `vectorspace_hip.h`: These are C++ templated classes for managing and manipulating vectors on GPU, including allocation, copying, and synchronization operations.

Next, we focus on the makefile for compiling the `qsim` backend. We modify the makefile to include compilation instructions for the HIP interface. This modification facilitated the build process for AMD GPU support, making integrating the newly developed backend into the `qsim` framework easy. We encountered a minor issue related to warp-level collective functions during the implementation. In CUDA, warp-level collective functions typically operate with a warp size of 32, which aligns with NVIDIA GPU architectures. However, AMD GPUs utilize a warp size of 64. To ensure seamless collective operations on AMD GPUs, we make a minor change in the code by ensuring the warp-level collective functions support a warp size 64.

4 EXPERIMENTAL SETUP

For our experiments, we use a computational node within the AMD GPU partition of the Dardel HPE Cray EX supercomputer at KTH, comprising a total of 56 GPU nodes. Each GPU computing node has an AMD EPYC 7A53 Trento processor featuring 64 cores and four AMD Instinct MI250X GPUs. Most importantly, each AMD Instinct MI250X GPU incorporates two Graphics Compute Dies (GCDs), presenting itself effectively as two distinct GPUs to programmers. This configuration mirrors the hardware setup of prominent supercomputers such as the current June 2023 [1], no.1 Frontier, and no.3 LUMI supercomputers, making our research pertinent to the execution of `qsim` on today's most advanced AMD GPU-based supercomputers. Furthermore, we employ an additional workstation with an Nvidia A100 to facilitate performance comparisons, particularly with `qsim` running on Nvidia GPUs utilizing the CUDA and cuQuantum backends. This supplementary setup features an AMD EPYC CPU complemented and one Nvidia A100 GPU. A comprehensive overview of our experimental setup, encompassing both hardware and software particulars, can be found in Table 1.

For assessing the performance of `qsim` on the CPU system, we use 128 OpenMP threads (two threads per core provided the best performance on the AMD CPU). In terms of GPU execution configuration, we assign 32 threads per block (referred to as 'threads per workgroup' using HIP terminology) for `ApplyGateL_Kernel` kernel and 64 threads per block for `ApplyGateL_Kernel`. These parameters are fixed as they correspond to the size of the shared memory arrays. It is important to note that opting for 32 threads per block in the `ApplyGateL_Kernel` kernel may lead to suboptimal GPU utilization, considering that the AMD wavefront (akin to warp in CUDA terminology) consists of 64 threads, as opposed to 32 in CUDA. In contrast, employing 64 threads per block in `ApplyGateL_Kernel` necessitates a significant algorithmic overhaul, as this straightforward extension surpasses the available shared memory capacity on AMD GPUs.

For test purposes, we use the Random Quantum Circuit (RQC) sampling circuit to evaluate the performance of our HIP backend. The RQC sampling is a key benchmark to demonstrate quantum computational supremacy, as sampling from the output distribution of large random quantum circuits surpasses the capabilities of classical computers [3]. We employ RQC circuits with 30 qubits for our evaluation. The RQC circuit definition involves randomly selecting gates, including single-qubit and two-qubit gates, and applying them to a predefined set of qubits [19]. To generate the circuits for our experiments, we utilize the Google Cirq library [20], as previously employed in the seminal work [3] and use the already available input file in the `qsim` GitHub repository [2].

Table 1: Hardware and software setup.

Setup	Details
CPU	AMD 7A53 Trento
Cores	64
Clock frequency	2.75 GHz (base)
Memory	512 GB DDR4
AMD GPU (# GCD)	AMD MI250X (2)
Memory per GCD	128 GB HBM2
Theoretical peak memory BW per GCD	1638.4 GiB/s
Theoretical peak SP FLOPs per GCD	23.95 TFLOP/s
Nvidia GPU	Nvidia A100
Memory per GPU	40 GB HBM2
Theoretical peak memory BW per GPU	1448 GiB/s
Theoretical peak SP FLOPs per GPU	10.5 TFLOP/s
`qsim`	0.16.3
Compiler	GCC 8.5.0
ROCm	5.3.3
CUDA Toolkit	CUDA 11.5
cuQuantum	23.03.0

Unless differently stated, all the calculations are performed in single precision, and we repeat the performance test five times and report the average values. The main performance metric is the execution time, which also includes the time to perform the gate fusion step. However, we found that gate fusion only took a small fraction of the total execution time ($< 2\%$). The standard deviations of these measurements are negligible, less than 1% For this reason, we omit the error bar in the plots.

Incorporating the HIP framework into the `qsim` backend offers more than just compatibility with AMD GPUs; it also brings forth valuable advantages, notably in profiling and tracing capabilities facilitated by the `rocprof` tool. These features provide an understanding of the performance and operational characteristics of the `qsim` state vector quantum computer simulator during its execution, generating a JSON file containing all the tracing data. Subsequently, this JSON file can be visualized using the `Perfetto UI` visualization tool [3] directly within a web browser. In Figures 1 and 6, we present two snapshots of the tracing information obtained through the `rocprof` tool. These figures prominently showcase the primary kernels, `ApplyGateH_Kernel` and `ApplyGateL_Kernel`, which play a central role in driving the execution of the `qsim` backend implementation. Additionally, they demonstrate the the use of asynchronous memory copies, an important optimization technique that enables the concurrent execution of computation and data movement.

[1]https://www.top500.org/lists/top500/2023/06/

[2]https://github.com/smarkidis/qsim/blob/hip_dev/circuits/circuit_q30

[3]https://ui.perfetto.dev/

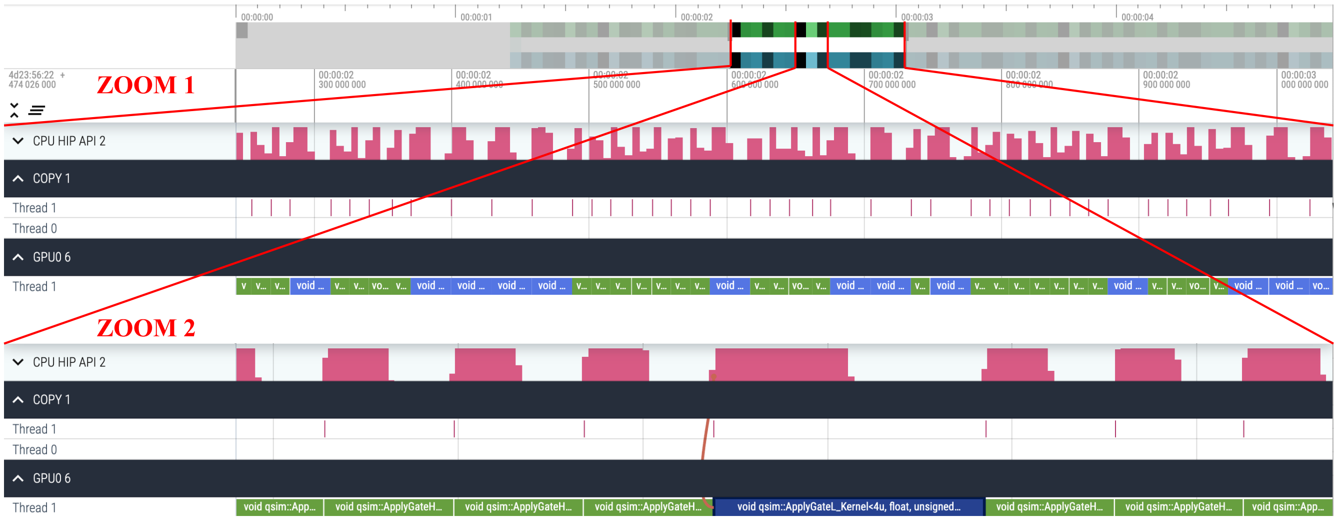


Figure 6: Tracing of the `qsim` HIP backend using the `rocprof` tool together with the `Perfetto UI` tool for the visualization. Two zoom regions are presented: the pink color represents the kernel execution. The `ApplyGateL_Kernel` kernel takes more time than the simpler `ApplyGateH_Kernel`.

5 RESULTS

In this section, we present the outcomes of our performance evaluation, wherein we compare the execution times of the `qsim` AMD CPU, AMD HIP, and CUDA backends while simulating the RQC with 30 qubits.

Our first test assesses the performance speed-up achieved by employing the AMD MI2050X GPU in contrast to using solely the on-node AMD Trento CPU. Figure 7 represents the execution time in seconds for simulating the 30-qubit RQC on the y-axis and the maximum number of fused gates on the x-axis. Notably, the quantity of fused gates has a critical impact on the AMD GPU and CPU performance. In our assessments, the fusion of four gates consistently yielded the optimal performance, whereas employing lower or higher numbers of fused gates led to suboptimal outcomes. Across all our tests, the AMD MI2050X GPU consistently outperformed the AMD EPYC Trento CPU by a substantial margin, achieving speeds up to seven to nine times faster. This demonstrates a significant performance gain attributed to the utilization of the AMD GPU.

The `qsim` simulator offers the flexibility to support both single- and double-precision calculations, albeit requiring separate compilations for each. Figure 8 illustrates the execution times of `qsim` runs, comparing double- and single-precision variables involved in the computation. Notably, calculations performed in double-precision exhibit an approximate slowdown of 1.8 to 2 times compared to those in single-precision. Upon examination of the state vector results, no substantial disparities were observed. This finding suggests that, at least for the RQC problem, single-precision calculations provide a convenient and accurate simulation while conserving half the memory compared to double-precision calculations.

In this final assessment, we aim to compare the performance of `qsim` when utilizing the CUDA and cuState (cuState is the state vector library in cuQuantum) backends on the Nvidia A100 GPU against the newly developed `qsim` HIP backend designed for AMD GPUs. Figure 9 represents the execution times in seconds for the HIP backend on AMD GPUs and the CUDA and cuState backends,

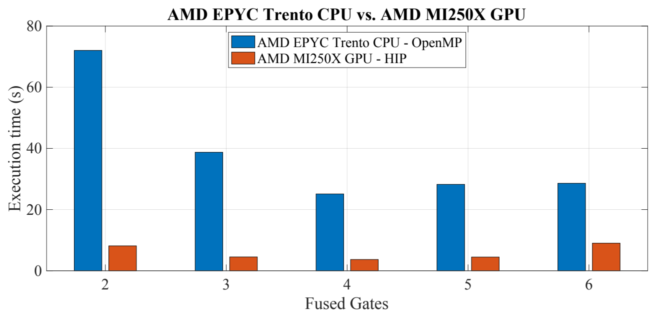


Figure 7: Execution time of the `qsim` state vector simulator on AMD Trento CPU and AMD MI250X GPU varying the number of fused gates.

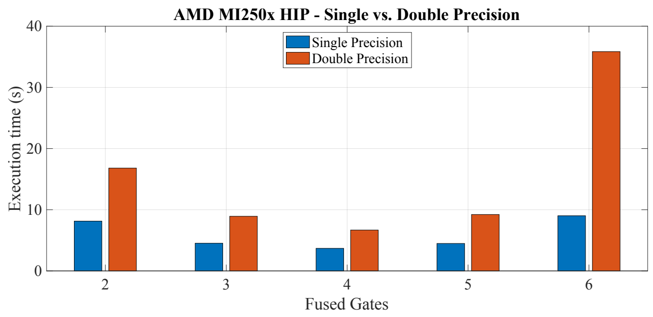


Figure 8: Difference in performance between using single and double precision with the `qsim` HIP backend on the AMD MI250X GPU.

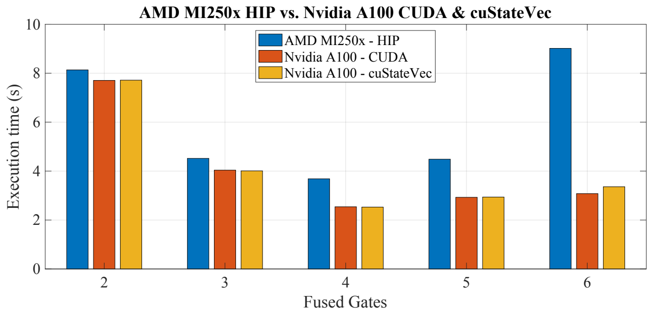


Figure 9: Execution time of the qsim CUDA, cuQuantum, and HIP backend on the Nvidia A100 and MI250x, varying the number of fused gates.

with variations in the maximum number of fused gates along the x-axis. Examining the performance of the CUDA and cuQuantum backends, we observe a clear dependency on the maximum number of fused gates, with the setup involving four fused gates consistently delivering the best performance. The distinction in performance between the CUDA and cuQuantum backends remains relatively modest, with a performance difference of less than 10%, favoring cuQuantum by a slight margin. This outcome underscores the high level of optimization already achieved in the CUDA backend, which has been successfully ported to HIP, making it a strong contender in terms of performance. It is noteworthy that the qsim cuQuantum backend supports multi-GPU setups, whereas the current CUDA backend does not offer this capability.

The most important insight derived from Figure 9 emerges from the comparison of qsim's performance on the AMD MI250X and Nvidia A100 GPUs. Across all our tests, qsim's performance on the A100 consistently surpasses that on the AMD MI250X, with the performance gaps becoming more pronounced with optimal gate fusion setups. For instance, with a two-gate fusion configuration, there is a 5% difference, while this disparity widens to 44% percent with a four-gate fusion setup. We also note that HIP backend performance deteriorates with larger gate fusion numbers, while this does not occur for the backend for Nvidia GPUs.

Our results underscore the significant performance gains achievable through GPU utilization, whether it is Nvidia or AMD GPU hardware. All GPU backends demonstrate a substantial performance advantage over the CPU backend, affirming the potential of GPU-based quantum simulation for handling large-scale quantum circuits Despite a slight lag behind the CUDA backend, the AMD HIP backend presents itself as a competitive alternative.

6 RELATED WORK

State vector quantum computer simulators have a rich history, evolving alongside the advancements in quantum computing technologies. An early milestone in parallel HPC quantum state vector simulation can be attributed to Obenland and Despain's pioneering work in 1997 [35]. In 2007, Da Raedt et al. [9] enabled the simulation of up to 36 qubits using the state vector representation, harnessing the power of up to 4,096 CPUs on the IBM BlueGene/L supercomputer. This marked one of the early efforts in the domain. More recently, Da Raedt et al. revisited their work from 2007 [8], highlighting the importance of HPC in quantum computer simulations.

As supercomputers have evolved, incorporating GPUs into their architecture has become imperative. It is important that quantum computer simulators adapt and support accelerators for efficient execution. In response to this need, Doi et al.[13] developed an Nvidia GPU backend, leveraging the Thrust library. This simulator is seamlessly integrated with the Qiskit Aer framework. Subsequent work by Doi et al.[12] introduced cache-blocking techniques, aligning with the methodologies utilized in Qiskit for multi-GPU acceleration. Furthermore, they further increased optimization efforts, focusing on multi-shot quantum simulations on GPUs [11]. The Quantum Exact Simulation Toolkit (QuEST) is another prominent HPC quantum computer simulator that extends support to Nvidia GPUs. Faj et al. investigated the advantages of utilizing Nvidia GPUs in Qiskit Aer, unveiling more than 40× performance gains with respect to execution on CPU. Their research demonstrated the substantial benefits derived from GPU utilization, particularly in the simulation of circuits with a relatively large number of qubits. Lastly, other emerging accelerators, such as FPGAs and streaming quantum languages [33], have also been investigated.

7 DISCUSSION AND CONCLUSION

Quantum computer simulators play a critical role in supporting the development and validation of quantum algorithms. In this study, we primarily focused on porting qsim, a well-established quantum computer simulator, to AMD GPUs. We leveraged the existing qsim CUDA backend and used the HIPIFY tool to provide a qsim HIP backend tailored for AMD GPUs. Our performance analysis centered on evaluating the HIP backend's capabilities, which we executed on a computing node equipped with the AMD MI250X GPU and the Trento AMD CPU. We used the RQC sampling benchmark, employing a circuit featuring 30 qubits. The results were striking, with the GPU outperforming the CPU by a remarkable margin, achieving seven to nine times faster speeds. Additionally, we explored the impact of double precision and observed that, while providing no discernible accuracy improvement, it incurred a performance penalty, slowing execution by a factor of 1.8 to 2 times. Our investigation also compared qsim's performance on the Nvidia A100 and AMD MI250X GPUs. The A100 GPU consistently outperformed the AMD counterpart, and this performance gap further widened with optimal gate fusion configurations. For instance, a two-gate fusion configuration exhibited a relatively 5% difference, whereas a four-gate fusion setup revealed a large 44% performance gap.

While our performance evaluation yielded promising results, it is essential to acknowledge that the HIP backend's performance still trails that of the Nvidia GPU backend. This disparity is likely attributable to the absence of fine-tuned optimization for the AMD GPU, a crucial step for future development. Optimization efforts should be supported through the analysis of tracing and profiling data gleaned through HIP tools. One such potential optimization lies in the potential of tensor cores [29] and matrix engines in both Nvidia and AMD GPUs, awaiting investigation. This optimization, however, necessitates a comprehensive understanding of the impact of lower mixed precision on quantum computer simulator calculations. In tandem with performance optimization, the multi-GPU porting for the HIP backend is an important goal for future work, offering the prospect of simulating quantum qubits with even larger qubit counts.

In summary, our work highlights the substantial performance advantage of GPU-based quantum computer simulation over traditional CPU approaches, underscoring its potential in handling complex, large-scale quantum circuits. Despite a performance lag compared to the CUDA backend, the AMD HIP backend emerges as a competitive alternative, poised for further optimization.

ACKNOWLEDGMENTS

This work is funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Finland, Germany, Greece, France, Slovenia, Spain, and the Czech Republic under grant agreement No 101093261. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) at KTH, partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

REFERENCES

[1] Yuri Alekseev. 2018. *Evaluation of the intel-QS performance on theta supercomputer.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[2] Thomas Alexander, Naoki Kanazawa, Daniel J Egger, Lauren Capelluto, Christopher J Wood, Ali Javadi-Abhari, and David C McKay. 2020. Qiskit pulse: programming quantum computers through the cloud with pulses. *Quantum Science and Technology* 5, 4 (2020), 044006.

[3] 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.

[4] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

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

[7] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 73–84.

[8] 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. https://doi.org/10.1016/j.cpc.2018.11.005

[9] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, and N. Ito. 2007. Massively parallel quantum computer simulator. *Computer Physics Communications* 176, 2 (2007), 121–136. https://doi.org/10.1016/j.cpc.2006.08.007

[10] Cirq Developers. 2023. *Cirq.* https://doi.org/10.5281/zenodo.8161252

[11] Jun Doi and et. al. 2023. Efficient Techniques to GPU Accelerations of Multi-Shot Quantum Computing Simulations. *arXiv preprint arXiv:2308.03399* (2023).

[12] Jun Doi and Hiroshi Horii. 2020. Cache blocking technique to large scale quantum computing simulation on supercomputers. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 212–222.

[13] Jun Doi, Hitomi Takahashi, Rudy Raymond, Takashi Imamichi, and Hiroshi Horii. 2019. Quantum Computing Simulator on a Heterogenous HPC System. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM, 85–93.

[14] Jennifer Faj, Ivy Peng, Jacob Wahlgren, and Stefano Markidis. 2023. Quantum Computer Simulations at Warp Speed: Assessing the Impact of GPU Acceleration. *arXiv preprint arXiv:2307.14860* (2023).

[15] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.

[16] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433.

[17] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa, and Nicolas PD Sawaya. 2020. Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits. *Quantum Science and Technology* 5, 3 (2020), 034007.

[18] 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.

[19] Jack D Hidary. 2019. *Quantum computing: an applied approach*. Vol. 1. Springer.

[20] Sergei V Isakov, Dvir Kafri, Orion Martin, Catherine Vollgraff Heidweiller, Wojciech Mruczkiewicz, Matthew P Harrigan, Nicholas C Rubin, Ross Thomson, Michael Broughton, Kevin Kissell, et al. 2021. Simulations of quantum circuits with approximate noise using qsim and Cirq. *arXiv preprint arXiv:2111.02396* (2021).

[21] Eric R Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia. 2019. *Programming quantum computers: essential algorithms and code samples*. O'Reilly Media.

[22] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific reports* 9, 1 (2019), 10736.

[23] Jin-Sung Kim, Alex McCaskey, Bettina Heim, Manish Modani, Sam Stanwyck, and Timothy Costa. 2023. CUDA Quantum: The Platform for Integrated Quantum-Classical Computing. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–4.

[24] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters. In *Sc20: international conference for high performance computing, networking, storage and analysis*. IEEE, 1–15.

[25] Boxi Li, Shahnawaz Ahmed, Sidhant Saraogi, Neill Lambert, Franco Nori, Alexander Pitchford, and Nathan Shammah. 2022. Pulse-level noisy quantum circuits with QuTiP. *Quantum* 6 (2022), 630.

[26] Salvatore Mandrà, Jeffrey Marshall, Eleanor G Rieffel, and Rupak Biswas. 2021. HybridQ: A hybrid simulator for quantum circuits. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. IEEE, 99–109.

[27] Stefano Markidis. 2022. On physics-informed neural networks for quantum computers. *Frontiers in Applied Mathematics and Statistics* 8 (2022), 1036711.

[28] Stefano Markidis. 2023. Programming Quantum Neural Networks on NISQ Systems: An Overview of Technologies and Methodologies. *Entropy* 25, 4 (2023), 694.

[29] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[30] Igor L Markov and Yaoyun Shi. 2008. Simulating quantum computation by contracting tensor networks. *SIAM J. Comput.* 38, 3 (2008), 963–981.

[31] Alexander McCaskey, Eugene Dumitrescu, Mengsu Chen, Dmitry Lyakh, and Travis Humble. 2018. Validating quantum-classical programming models with tensor network simulations. *PloS one* 13, 12 (2018), e0206704.

[32] Jarrod R McClean, Nicholas C Rubin, Kevin J Sung, Ian D Kivlichan, Xavier Bonet-Monroig, Yudong Cao, Chengyu Dai, E Schuyler Fried, Craig Gidney, Brendan Gimby, et al. 2020. OpenFermion: the electronic structure package for quantum computers. *Quantum Science and Technology* 5, 3 (2020), 034014.

[33] Gilbert Netzer and Stefano Markidis. 2023. QHDL: a Low-Level Circuit Description Language for Quantum Computing. *arXiv preprint arXiv:2305.09419* (2023).

[34] Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information.

[35] Kevin M Obenland and Alvin M Despain. 1998. A parallel quantum computer simulator. *arXiv preprint quant-ph/9804039* (1998).

[36] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 4213.

[37] Qiskit contributors. 2023. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

[38] Quantum AI team and collaborators. 2020. *ReCirq.* https://doi.org/10.5281/zenodo.4091470

[39] Quantum AI team and collaborators. 2020. *qsim.* https://doi.org/10.5281/zenodo.4023103

[40] The cuQuantum development team. [n. d.]. *NVIDIA cuQuantum SDK.* https://doi.org/10.5281/zenodo.6385574

[41] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. 2019. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Information* 5, 1 (2019), 86.