Randomized Benchmarking of Quantum Gates on a GPU

Syed Zawad[†], Feng Yan[†], Rui Wu[§], Lee Barford[†‡], Frederick C Harris, Jr. [†]

[†]*Dept of Computer Science and Engineering*
University of Nevada
Reno, USA

[§]*Department of Computer Science*
East Carolina University
Greenville, NC

[‡]*Keysight Laboratories*
Keysight Technologies
Reno, NV

szawad@nevada.unr.edu fyan@unr.edu wur18@ecu.edu lee.barford@ieee.org fred.harris@cse.unr.edu

***Abstract*—While quantum computing has shown great promise in the field of computer science, a lack of actual practical quantum hardware means that mainstream research must rely on simulations. As such, a wide number of quantum computing simulation libraries have been developed, each with their own strengths and weaknesses. A good simulator must not just be accurate, but fast as well. This is especially relevant for quantum systems since the problem size growth for quantum systems is super-exponential. For this paper, we introduce a quantum computing simulation system that takes advantage of multiple gpus to achieve up to 400 times faster simulation time. We discuss the implementation details and provide analysis of its performance. We also demonstrate how the real-world phenomenon of quantum gate incoherence can be accurately simulated by varying the floating point precision and demonstrate it by using a precision of 9 bits, which we evaluate using Randomized Benchmarking.**

Keywords—quantum computing, randomized benchmarking, GPU, performance analysis

I. Introduction

The field of quantum information explores the possibilities of exploiting the laws of quantum mechanics to gain benefits in computational complexities that are otherwise largely problematic for classical computers to solve. Quantum computing involves using the superposition principle to carry out computational tasks in a more efficient way than is possible with devices governed by classical physics. There is a broad range of longstanding problems in strongly correlated systems, and quantum computing has shown great prospect in solving them. This has encouraged a significant increase in research, especially over the last decade. The tools required to design, build, and implement these quantum systems [1, 2] have seen rapid development and are still being developed, currently reaching some very sophisticated levels [3]. Research breakthroughs in ultra-cold atoms and photons have become more commonplace.

However, building quantum computers represents an immense technological challenge and, at present, the quantum hardware is only available in research labs. Under these circumstances quantum simulators have become valuable instruments in developing and testing quantum algorithms and in the simulation of physical models used in the implementation of a quantum processor. Simulating a quantum computer on a classical computer is a computationally hard problem. According to Feynman's paper [1] classic computers will never be able to perform simulations of full behavior of a quantum system in a polynomial time. Because of the exponential behavior of quantum systems, simulating them on conventional computers requires an exponential amount of operations and storage. Parallelization alleviates this problem, allowing the simulation of more qubits at the same time or the same number of qubits to be simulated in less time.

This parallelism can be achieved through the use of Graphics Processing Units. Modern graphics processing units (GPUs) have been at the leading edge of increasing chip-level parallelism for some time. While originally designed to perform calculations for graphics in video editing and video games, they are now being widely used for general purpose programming in many fields which require intense parallel computations. For this paper, we demonstrate that GPUs are a viable candidate for simulating quantum computers in terms of both accuracy and speed of execution. We also show how the GPU accuracy changes with changes in its floating point precision, and how that can be used to simulate accurate models of decaying quantum gates. This paper makes three contributions -

- *Develop a GPU-based Quantum Simulation framework to accurately simulate the application of Quantum Gates* – The simulation uses the Clifford gates [5] model, and is implement in C and CUDA. It provides the linear algebra calculations required to model the fundamentals of the Clifford gates system. The framework has support for multiple GPUs.
- *Optimize quantum simulator calculations* - Certain linear algebra functions such as transpositions and matrix inner products can be considered redundant for some parts of the quantum computing model. This paper discusses how they can be removed or optimized to speed up the execution for quantum bit calculations as well as how and where to reduce the problem size.
- *Apply Randomized Benchmarking* – Randomized Benchmarking is currently one of the more popular tests which are used to analyze the fidelity of hardware quantum gates. It measures how accurate a certain hardware is in terms of performing quantum calculations. We apply this to our GPU implementation and show that GPUs can be an accurate simulator. We also vary the precision of the calculations manually and show how a varying precision can accurately model the gate incoherence present in real systems.

The rest of this paper is organized as follows: In Section 2 we talk about the background of our quantum model. We provide a short explanation of the randomized benchmarking algorithm and give a brief description of the NVidia CUDA architecture. In Section 3 we review the current state of research in the application of quantum computing simulations for GPUs and explain how our research fits there. In Section 4 we discuss our implementations details, our optimizations and parallel strategies. For Section 5, we provide the results of our optimizations and parallel implementations. We then compare and discuss our observations, and draw our conclusions and present some future work in Section 6.

II. Background

A. Quantum Computing Model

The concept of using Quantum Computer simulations in a classical computer were an extension of Feynman [1]. The application of the principles of quantum physics in the computer area led to the concept of quantum computer, in which the data isn't stored in bits like in the conventional memory, but as a combined state of several systems with 2 qubit states. Nowadays, the most common model involves using Pauli matrices and Clifford groups to perform computations for quantum processes, consistent with the Gottesman – Knill theorem [6, 7].

The fundamental concept, or representation of a quantum system revolves around the *qubit*. The qubit can be considered as the equivalent bit representation of a quantum system. While a classical computer uses 0's or 1's to represent data, a quantum system uses the qubit. The qubit is represented as a square matrix of the dimensions 2^n-by-2^n, where n is the number of bits in the qubit. Fig. 1. Shows the density matrix of a 2-bit qubit.

$$\begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 0.1 & 0 & 0 & 0 \\ 01 & 0 & 0.2 & 0 & 0 \\ 10 & 0 & 0 & 0.3 & 0 \\ 11 & 0 & 0 & 0 & 0.4 \end{array}$$

Fig 1. A 2-bit qubit density matrix representation

Here, the rows and columns represent the possible bit combinations, and the corresponding diagonal values represent the probability with which the bit combination will be returned when the qubit state is read. The density matrix contains real and imaginary values. The density matrices must be have sum of diagonal elements to 1, must be a Hermetian and a positive semi-definite matrix.

Similarly, as with classical computers, quantum computers also have gates which transform the probabilities of the bit combinations in a qubit. They are also represented by 2-by-2 density matrices, and are complex and unitary. Fig. 2 shows the qubit representation of an *X* gate (also called the qubit *NOT* gate).

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Fig 2. X (qubit NOT) gate. It switches the probabilities between the bit combinations.

A 2-by-2 gate, however, can only be applied to a single qubit density matrix. In order to be applied to higher order matrices, the gate must be expanded to equal the dimensions of the qubit itself. This is done via creating the appropriate Kronecker product of the gate. The formula for creating an n-qubit gate is as follows –

$$\underbrace{\underbrace{I \otimes \ldots \otimes I}_{i-1 \text{ times}} \otimes G \otimes \underbrace{I \otimes \ldots \otimes I}_{n-i \text{ times}}}_{n \text{ times total}}$$

where i is the ith qubit that the gate will apply to, n is the number of qubits, I is a 2-by-2 identity matrix, G is the 2-by-2 gate matrix, and $\otimes$ represents the kronecker product function. The above equation results in a 2^n-by-2^n matrix which can be applied to the 2^n-by-2^n qubit density matrix. Application of qubit gates to qubits is done by –

$$Q_{\text{new}} = G^{\dagger} Q G$$

Here, Q_{new} is the newly transformed qubit resulting from the application of gate G to the original qubit Q. $G^{\dagger}$ is the Hermetian of the gate G.

B. Randomized Benchmarking

The purpose of the Randomized Benchmarking method is to find how accurate the computations are after applying the gates to a certain initial starting qubit. What it measures is the "proximity" of the result after the application of gates. This "proximity" can be considered as how erroneous the gates are and is known as the *infidelity*. Algorithm 1 shows the code for Randomized Benchmarking.

Algorithm 1: Randomized Benchmarking

```
Input:
Number of Runs $N$
Number of Gates $M$.
Initial Qubit $Qubit$
Output:
$Fidelity$
1  for $n$: 1 to $N$ do
2      for $m$: 1 to $M$ do
3          $G_m$ = getRandomGate();
4          $Qubit$ = applyGate($G_m$,$Qubit$);
5          $GateStack$.push($G_m$);
6      end
7      for $m$: $M$ to 1 do
8          $G_m$ = hermetian($GateStack$.pop());
9          $Qubit$ = applyGate($G_m$,$Qubit$);
10     end
11     $fidelity$ = calculateFidelity($Qubit$);
12 end
```

The algorithm takes in the number of test runs N, the number of gates to apply per run M and the initial qubit *Qubit* as inputs. For every run, M gates are chosen at random and applied one after the other. These gates are then stored in a LIFO queue. The initial Qubit undergoes a series of probability transformations due to the application of these gates that results in a completely different set of probabilities. After the first inner loop, the initial Qubit has been destroyed. The next inner loop pops the queues from the LIFO Queue and applies the Hermetian transformation to the gate. One of the gate properties is that the conjugate transpose of the gates is equal to itself, and that they are unitary gates. This means that by applying a gate to a Qubit and applying the Hermetian of the gate to the resulting qubit will result in the Qubit reverting back to its original state. As such, the application of a sequence of gates and then the Hermetians of those gates should ideally result in the initial Qubit. Practically, however, quantum hardware does not give the exact initial state. Due to noise in gate hardware, the exact calculations are rarely accurate, and this becomes apparent after the application of the gates and their Hermetians. It is expected that as more gates are applied, the "further" the final state is from the initial state.

C. NVidia CUDA and GPUs

The Compute Unified Device Architecture (CUDA) library, developed by NVIDIA, is a software and hardware architecture that enables the users to harness the high counts of parallel processing power of the recent NVIDIA graphics cards. From the hardware perspective, the GPU consists of several multiprocessors working in a SIMT (Single Instruction Multiple Thread) fashion, each of them containing a certain number of streaming processors. In order to develop GPU-enabled applications, programmers can make use of various programming languages: C/C++ for CUDA, OpenCL, Fortran or DirectCompute. However, CUDA is the proprietary library provided by the hardware developers themselves, and thus provides many functionalities that allow users to fully utilize the hardware. The primary difference between a GPU and a CPU is that GPUs contain a high number of less powerful cores while CPUs contain a few number of highly powerful cores. Other than that, the other properties are similar. GPUs contain its own memory spaces; (1) Global memory - Data stored in global memory is visible to all threads within the application (including the host), and lasts for the duration of the host allocation. (2) Shared Memory – Data stored here is visible to all threads within the block that allocates it. This type of memory allows for inter-thread communication to occur and permits the sharing of data between threads. This is also faster than the global memory. (3) Local memory – Local memory has the same properties as normal registers, but has a larger available memory size but it is much slower. Apart from these, there are also Texture and Constant memory available, but it is not relevant in this context. CUDA provides a hierarchical execution model for execution of its threads. The abstraction is done at 4 levels. These abstractions are (in order of highest abstraction to the lowest) - grids, blocks, warps and threads

III. Related Work

The concept of Quantum Systems was first put forth by Feynmen [1]. The paper emphasized the complexity of simulating quantum systems using classical computers. A well-controlled system can be built from the bottom up, and by doing so, one could create a computer whose constituent parts are governed by quantum dynamics generated by a desired Hamiltonian. However, at that time period, the computational power required to even describe the quantum system which scales exponentially with the number of its qubits was practically infeasible. Additionally, delving this deeply into the properties of this system led to the discovery of difficult to compute properties of a quantum many-body model, such as the nature of quantum-phase diagrams. This initial proposed model is known as the "quantum simulator".

Since then, there have been great strides towards feasible quantum computers, even though much work is yet to be done to make quantum computers mainstream. As such, quantum computing simulation libraries are still what drives the majority of research. These simulators come in different variations and levels of complexity [8], each catering to a specific need. Sequential quantum simulators are many, and have a good variety of representations and contains different types of simulators [9]: quantum programming languages (QCL, Q language, Quantum Superpositions, QuBit), quantum compilers (Qubiter, GQC), quantum circuits simulators (QCAD, QuaSi, Libquantum), quantum hardware emulators (QCE, QSS) and purely pedagogical software (quantum Turing machine simulator, Quantum Search Simulator, Shor's algorithm simulator). The need for parallel simulators emerges due to the super-exponential nature of quantum computation. It is extremely time consuming for classical computing devices simulate it. The first parallel simulator was developed by Obenland and Despain [10], but was based on the physical model of a very specific model and so did not see mainstream use. Since then, there have been many parallel CPU implementations, [10, 11]. However, they all fall short due to lack of scalability. The latest approach that researchers are taking to further optimize the simulators are through the GPUs. Much work has already been done on simulating specifics applications [12, 13]. Similarly, several generic single GPU and distributed GPU systems for quantum simulators have been developed and see widespread use [14, 15].

One of the key problems with quantum systems are that they have degrading accuracy [16, 17]. While many companies are racing to become the first to develop the first real quantum computer, one of the major hurdles of noisy systems still exist. To facilitate the work in this field, researchers have come up with standards to measure and standardize the testing process [18, 19, 20, 21, 22]. The industry standard so far has been QST and GST, but [23] argues that they are both too slow and bad at scaling. To counter this, they propose a new type of testing called Randomized Benchmarking. Since then, there have been quite a few experiments where RB has been used to good effect [23, 24]. The scalability, simplicity and ease of implementation have made RB one of the currently favored testing methods for quantum hardware. While much work regarding quantum simulators have been done, there has been no research into how RB will perform on GPUs. Since GPUs are now the go-to implementations for quantum simulators, we need to find a means of simulating the deteriorating effect of applying gates to qubits.

IV. Implementation

One of our key motivations is to be able to generate the decaying effect shown by real quantum gates on the GPU. We do this by manipulating the floating point precision of the calculations of the operations. We do this via the formula–

$$R_{factor} = 1 \ll M$$

$$Value = \frac{round(Value_{original} * R_{factor})}{R_{factor}}$$

Here, M is the precision in bits, and $Value_{original}$ is the actual value with full 64-bit precision. This function is applied to all mathematical calculations done in every thread. For our implementation, we start by setting the probability of all qubits being 0 to 1.0. In other words, our $Q_{old}[0][0]$ is set to 1.0. With full precision, we expect that the value for $Q_{old}[0][0]$ will be 1.0 after going through Randomized Benchmarking. With less precision, we will find that the value will deviate from 1.0. *Fidelity* is the measure of how much the final value deviates. This is given by –

$$Fidelity = 1 - conj(Q[0][0]) * Q[0][0]$$

Q is the final qubit after applying Randomized Benchmarking. The closer *Fidelity* is to 1.0, the less the noise is in the system. For our experiments, we vary the precision and measure the corresponding *Fidelity* with a range of gates. Next, we talk about the two facets to the implementation of our framework. First we focus on reducing the number of computations for the operations while maintaining the accuracy of the results. Then we focus on how to implement a parallelized version of the algorithm for deployment on the GPU.

A. *Optimizations*

For the linear algebra operations involved in applying the gates to the quantum bit and for calculating the Hermetians, the number of calculations were reduced by taking advantage of the matrix properties of the gates and the qubits. These properties enable the reduction of the problem size due to rendering most of the computations necessary redundant. Specifically, three major algorithmic changes were done.

The first change was to only calculate the Hermetian of gates (line 8 of Algorithm 1) for specific gates. The traditional way of applying gates in quantum simulators use the transposed conjugate of the gate matrix and the gate matrix itself. The original qubit is then right multiplied by the transpose-conjugate and left multiplied by the original gate. These inner products are done in sequence to each other. It should also be noted that for multi-qubit systems the multi-qubit gate is first generated by the Kronecker product and then the conjugate transpose is applied. However, this conjugate transpose step can be completely ignored due to the fact that gate matrices must be Hermetian matrices. In other words, one of the properties that a matrix must fulfill to be a gate is that it must be a conjugate transpose of itself. Therefore, the Clifford group of gates (*X, Y, H, Z*) used are all Hermetians, meaning that they do are not required to undergo the Hermetian transformation. This change effectively removes one full computational step of the O(2^{2n+1}), where n is the number of qubits. The second optimization done involves using the Eigen property of the gates that reduces the computations to only using the upper half of the square matrices. Algorithm 1 can be unwound in the form of the equation –

$$G_{right} = G_1 G_2 \ldots G_{N-1} G_N G_N^H G_{N-1}^H \ldots G_2^H G_1^H$$

$$G_{left} = G_1 G_2 \ldots G_{N-1} G_N G_N^H G_{N-1}^H \ldots G_2^H G_1^H$$

$$Q_{new} = G_{right} Q_{old} G_{left}$$

where N is the total number of gates. However, we know that the Qubit Q_{old} can be decomposed as –

$$Q_{old} = LL^T$$

where L is the Eigen decomposition of Q_{old}. So now we can reduce the original equations to

$$G_{right} = G_1 G_2 \ldots G_{N-1} G_N G_N^H G_{N-1}^H \ldots G_2^H G_1^H L$$

$$G_{left} = G_1 G_2 \ldots G_{N-1} G_N G_N^H G_{N-1}^H \ldots G_2^H G_1^H L^T$$

$$Q_{new} = G_{right} G_{left}$$

When calculating the infidelity, we are only interested in the first diagonal value of the resulting matrix, i.e. Q_{new}[0][0]. Therefore, instead of calculating the full matrix, we may simply get the inner product of the first column of G_{right} and the first row of G_{left}. Thus, the total number of calculations of the full algorithm sequence can effectively be reduce to half, which contributes greatly to the reduction of the execution time of the complete run.

The last and most impactful of the optimizations is the complete removal of the calculation of the Kronecker product. The calculation of the Kronecker product takes place in line 3 of Algorithm 1. While getting the random gate, a random qubit is chosen from the *N* qubits. Then a random gate from among the Clifford groups is chosen and then the appropriate number of Kronecker multiplications are done to get the full *N*-qubit gate. Generating this gate for every step of the algorithm is time consuming, and any reduction here will reduce the total execution time greatly. The eventual product of the Kronecker multiplications is a full *n*-qubit gate, which is then multiplied against the qubit matrix. However, the *n*-qubit gate is very sparse, meaning that most of the values need not be generated via Kronecker at all. Additionally, given the index, the exact value of the Kronecker product can be deduced due to the structured expansion that Kronecker multiplications result in.

B. *CUDA Implementation*

As shown in Algorithm 1, there is one outer loop and two inner loops. The inner loops are involved in the actual calculations while the outer loop is based on the number of runs. The two inner loops contain the inner product of the matrices. However, in order for the calculations to be accurate, these operations must be applied in sequence. Therefore, the scope for parallelization is within the operation calculation itself. The operations to be parallelized here are the Hermetian, the optimized Kronecker products (*n*-qubit gate generation) and the inner product of the qubits and the gate matrices. The memory was allocated using CUDA's unified memory.

For parallelizing the inner product for this simulation scheme, the qubit state matrix is partitioned in sets of fixed dimensions and assigned to CUDA blocks, where the sub-vectors are processed in parallel on a SIMT (single Instruction Multiple Thread) fashion. The index multiplications are done in parallel threads. The final sum is done by reduction, where every summation between two values are done in parallel. The major bottleneck here is during the synchronization of the threads; every sum step needs a blocking call for the previous summing step so that the values required for the sum have been correctly calculated. For a matrix inner product of two square matrices of the dimensions 2^n-by-2^n, we will end up with a new matrix of dimensions 2^n-by-2^n. For maximum parallelism, every index is calculated in a separate independent thread. The multiplication and reduction are also done in independent threads. During deployment of the threads, the number of threads per CUDA block was kept at the maximum possible number of threads (i.e. 1024) for the GPU experimented on. This is not enough for large qubit sizes, so the number of blocks used were set such that –

$$number\ of\ blocks = \frac{2^{2n}}{1024}$$

This ensures that there are always enough parallel threads to get the maximum parallelism possible. There were also checks in place to ensure that the number of threads were multiples of 32 in order to keep the number of threads consistent with the warp sizes.

For the Hermetian calculations, the conjugate and the transpose functions on a single matrix index were merged to form a single operation where given an index, the transpose of the index is calculated by getting the value of the "mirror" index and then the conjugate of that value calculated and stored. Each index is run in a separate thread, with the threads distributed in the same manner as for the inner product. The dynamic qubit gate generation without the use of Kronecker products is also done on the GPU. The gate is generated and stored in the GPU's global memory, so there is no need to transfer the large gates from the host to the device and vice versa. The function `applyGate()` from Algorithm 1 deploys a single thread for the each of the 2^{2n} values in the matrix. This is run once when a random gate is generated (line 3 Algorithm 1). The execution time for the dynamically generated gate instead of doing the Kronecker products is further beneficial since the Kronecker product calculations, had they been done on the GPU, would have required thread synchronization for every loop iteration. However, the dynamic generation requires no such bottleneck and the gates require much less time to be generated. The multi-GPU implementation is done for the outer loops. Note that the final value for fidelity is calculated for every run of the outer loop, and is independent of the rest of the other calculations. Therefore, the outer loop can also be deployed in parallel threads. We take advantage of this fact by deploying the different outer loops in different GPUs. The devices never need to be synchronized.

V. Experiment and Results

The test bed used for the experiments contain 4 NVidia GTX 1080s. The CPU is Intel(R) Xeon(R) CPU E3-1225 v3, 3.20GHz with 4 Cores and has 64GB of RAM. The operating system used was Linux's Ubuntu 16.06. The timings were taken using CUDA's event synchronization library functions. The values used are means from 20 runs. Profiling was done using NVidia's profiler. Fig. 3 shows the pure execution time for the single GPU implementations, including the computation reduction optimizations. The log-scaled graph shows the clear benefits in performance of the GPU's parallelized version. We can see that for smaller number of qubits, the sequential version is better since the overhead of moving the qubit from the CPU to the GPU is too large to make up for the execution time reduction. However, at around the 5th qubit, the execution time of the sequential version starts taking more time than the parallel version. By the 13th qubit, the total speedup is around 400 times.

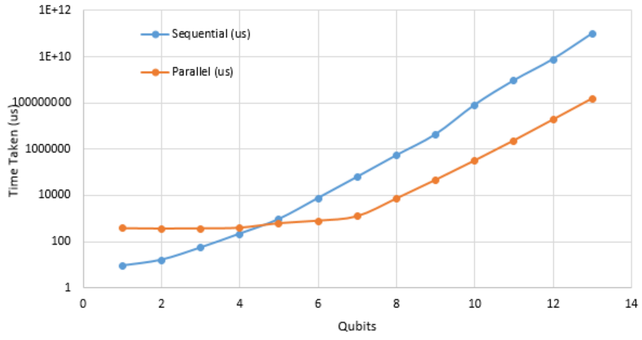

Figure 3. – Sequential versus Parallel execution times

The throughput graph (Fig. 4) shows that up to the 7th qubit there is a clear benefit from parallelism but plateaus out after that. The bulk of the computations among all the steps are taken up by the matrix multiplications. The synchronization required for the sum reduction becomes a significant bottleneck at that stage, resulting in a flat throughput. Fig. 5 presents the execution time of the multi-GPU implementation of the Randomized Benchmarking algorithm. As expected, the 2-GPU implementation is clearly almost twice as better as the single GPU case.

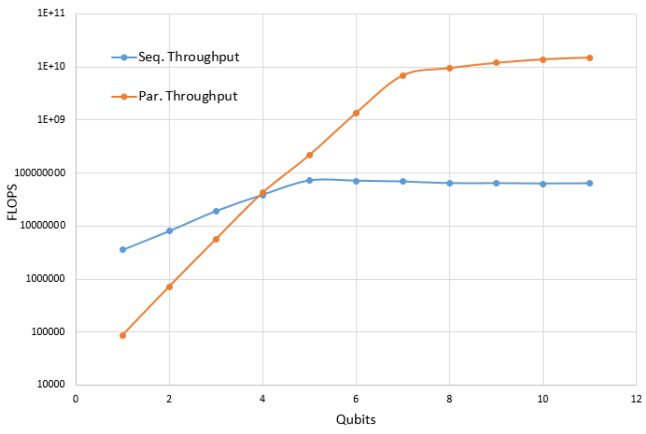

Figure 4. – Sequential versus Parallel throughput

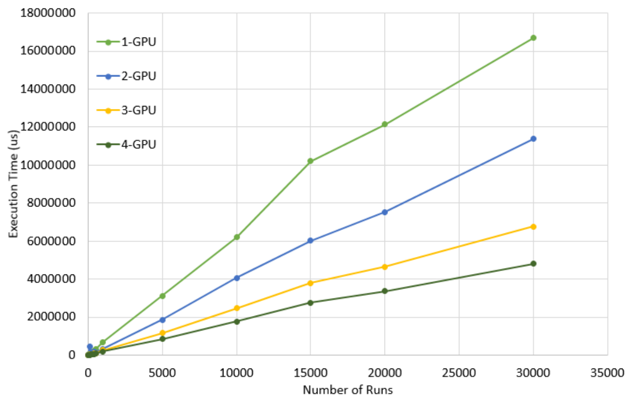

Figure 5. – Multi-GPU execution time

However, the addition of more GPUs do not clearly benefit at the same scale, as can be see for the 3 and 4 GPU cases. As the number of GPUs increase, the benefits of adding more GPUs tend to decrease. The overall execution time does decrease with increasing number of runs, and it scales linearly. Ideally, with 4 GPUs we should expect a speedup of 4 times if the scale was perfectly linear. The graphs show that this is not the case, and we have a speed up of around 3.6 times with 4 GPUs. This phenomenon can be explained by the fact that while pure execution time may speed up four-fold, the overall execution time given here also includes the data transfer time from host to device and vice versa, which does not scale linearly. Thus a sub-linear effect is introduced and the benefits of using multiple GPUs are somewhat diminished. Nonetheless, the speed up achieved is 90% of the maximum expected.

The final set of experiments performed focuses on the evaluation of the GPU's calculated fidelity based on the varying bit precision. The full implementation of the system was done using double precision. We have observed that the fidelity largely stays the same throughout the bit values between 64 and 10 bits, no matter how many gates are applied after each other. Once we started seeing decreasing fidelity, we varied the number of gates and the qubit size to understand how the fidelity changes against them. Fig. 6 shows how the fidelity is affected by an increasing number of gates. The data shown is the calculated mean and is using a precision of 9 bits. The trend here shows a strong inverse correlation between the increasing number of gates applied and the fidelity. This is consistent with the findings in [17, 19], where real quantum systems show a tendency to deteriorate with higher number of gates.

Another observation here is that for higher qubits, the fidelity deteriorates faster than for smaller qubit sizes. This is due to the fact that larger qubits undergo more computations and are thus more affected by round off errors occurring due to coarser precisions.

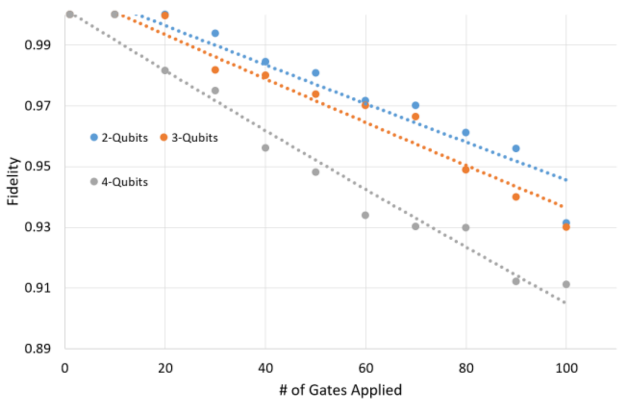

Figure 6. – Fidelity vs. Number of Gates Applied with varying qubits

VI. Conclusion

We introduce a set of new linear algebra optimizations that reduce the problem sizes of quantum computing calculations that drastically reduce execution time as well as make it easy to parallelize. These should be generic enough to be applicable for all quantum computing systems since these reductions were done at the most basic levels which are required by virtually all quantum simulators. We have conducted experiments based on which we can conclude that Quantum simulators benefit greatly from their usage of the parallelism afforded by multi-GPU systems in addition to the reduction in problems size, achieving a speedup of more than 400 times. Additionally, simply by changing the precision we were able to simulate the decaying effect of real quantum systems, making it a lightweight solution to a complicated simulation problem. While this is a sufficient solution currently, future work can involve usage of Gaussian noise to make gate decay simulation more faithful to how actual Quantum systems perform.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under grant number IIA-1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Feynman, R.P., Simulating physics with computers. *International journal of theoretical physics*, 21, no 6-7 (1982), pp.467-488.

[2] Buluta, Iulia, and Franco Nori. Quantum simulators. *Science*, 326, no. 5949 (2009): 108-111.

[3] Lin, Y-J., R. L. Compton, K. Jimenez-Garcia, J. V. Porto, and I. B. Spielman. Synthetic magnetic fields for ultracold neutral atoms. *Nature*, 462, no. 7273 (2009): 628-632.

[4] Cirac, J. Ignacio, and Peter Zoller. Goals and opportunities in quantum simulation. *Nature Physics*, 8, no. 4 (2012): 264.

[5] Bravyi, S., and A. Kitaev., Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A*, 71, no. 2 (2005): 022316.

[6] Gottesman, D., The Heisenberg Representation of Quantum Computers, in *Proceedings of the 22nd International Colloquium on Group Theoretical Methods in Physics*, pp 32-43, (International Press, Cambridge, 1999), p. 23

[7] Aaronson, S., and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70, no. 5 (2004): 052328.

[8] Eason, G., B. Noble, and I. N. Sneddon, On certain integrals of Lipschitz-Hankel type involving products of Bessel functions, *Phil. Trans. R. Soc. Lond. A*, (1955) 247, 529-551

[9] Quantiki. List of QC Simulators, 25 Apr. 2018, www.quantiki.org/wiki/list-qc-simulators.

[10] Obenland, K. M., and A. M. Despain. A parallel quantum computer simulator. *arXiv* preprint quant-ph/9804039 (1998).

[11] Brandl, M. F., A quantum von Neumann architecture for large-scale quantum computing in systems with long coherence times, such as trapped ions. *arXiv* preprint arXiv:1702.02583 (2017).

[12] Ufimtsev, I. S., and T. J. Martinez. Quantum chemistry on graphical processing units - Strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, 4, no. 2 (2008): 222-231.

[13] Maia, J. D., G. A. Urquiza Carvalho, C. P. Mangueira Jr, S. R. Santana, L. A. Cabral, and G. B. Rocha, GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations. *Journal of chemical theory and computation*, 8, no. 9 (2012): 3072-3081.

[14] Amariutei, A., and S. Caraiman, Parallel quantum computer simulation on the GPU. In *Proceedings of the 15th International Conference on System Theory, Control, and Computing (ICSTCC)*, 2011, pp. 1-6. IEEE.

[15] Gutierrez, E., S. Romero, M. A. Trenas, and E. L. Zapata. Parallel quantum computer simulation on the CUDA architecture. In: Bubak M., van Albada G.D., Dongarra J., Sloot P.M.A. (eds) *Computational Science – ICCS 2008. ICCS 2008. Lecture Notes in Computer Science*, vol 5101. pp. 700-709, Springer, Berlin, Heidelberg

[16] Deutsch, D., and R. Jozsa, Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A*, 439, no. 1907 (1992): 553-558.

[17] Shor, P. W., Scheme for reducing decoherence in quantum computer memory. *Physical review A*, 52, no. 4 (1995): R2493.

[18] O'Brien, J. L., G. J. Pryde, A. Gilchrist, D. F. V. James, N. K. Langford, T. C. Ralph, and A. G. White, Quantum process tomography of a controlled-NOT gate, *Physical review letters*, 93, no. 8 (2004): 080502.

[19] Knill, E., Quantum computing with realistically noisy devices. *Nature*, 434, no. 7029 (2005): 39.

[20] Reichardt, B. W. Quantum universality by state distillation, *arXiv* preprint quant-ph/0608085 (2006)

[21] Raussendorf, R., and J. Harrington, Fault-tolerant quantum computation with high threshold in two dimensions. *Physical review letters*, 98, no. 19 (2007): 190504.

[22] Mohseni, M., A. T. Rezakhani, and D. A. Lidar. Quantum-process tomography: Resource analysis of different strategies. *Physical Review A*, 77, no. 3 (2008): 032322.

[23] Knill, E, D. Leibfried, R. Reichle, J. Britton, R. B. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland, Randomized benchmarking of quantum gates. *Physical Review A*, 77, no. 1 (2008): 012307.

[24] Magesan, E., J. M. Gambetta, and J. Emerson. Scalable and robust randomized benchmarking of quantum processes, *Physical review letters*, 106, no. 18 (2011): 180504.

[25] Ryan, C. A., M. Laforest, and R. Laflamme. Randomized benchmarking of single-and multiqubit control in liquid-state NMR quantum information processing, *New Journal of Physics*, 11, no. 1 (2009): 013034.