Simulating Quantum Computers Using OpenCL

Adam Kelly

May 1, 2018

I present QCGPU, an open source Rust library for simulating quantum computers. QCGPU uses the OpenCL framework to enable acceleration by devices such as GPUs, FPGAs and DSPs. I perform a number of optimizations including parallelizing operations such as the application of gates and the calculation of various state probabilities for the purpose of measurement. Using an Amazon EC2 p3.2xLarge instance, the library is then benchmarked and also compared against some preexisting libraries with the same purpose. The presented library is limited only by the memory of the host machine or that of the device being used by OpenCL. The finished software is available at https://github.com/qcgpu/qcgpu-rust.

1 Introduction

Quantum computers are thought to be the key to some types of problems, such as factoring a semiprime integer [4][17], calculating discrete logarithms, the search for an element in an unstructured database [9][22], super dense coding [21], simulation of quantum systems, along with many other algorithms. Currently, the Quantum Algorithm Zoo, a website that details many algorithms for quantum computers cites 386 papers, at the time of writing [19]. It has also been suggested that quantum computers could create new opportunities in the fields of chemistry [12], optimization [14] and machine learning [16].

While it is not feasible to solve some of these problems on classical computers, the quantum algorithms do not violate the Church-Turing theorem and thus can be, to a small extent, simulated using classical computers.

There are some real quantum computers, such as IBM's quantum experience [6], which has semi-public access to a 5-qubit machine, a 16-qubit machine and a 20-qubit machine through their software library qiskit [1]. With devices now providing up to 20 controllable qubits, there are many issues being raised, including (most importantly) the ability to assess the correctness, performance and scalability of quantum algorithms.

It is this issue which simulators of quantum computers address. They allow the user to test quantum algorithms using a limited number of qubits and calculate measurements, state amplitudes and occasionally implement features which help in this testing process such as density matrices.

2 Background

2.1 Existing Research

There are many existing quantum computer simulators (many are listed at [2]) along with some existing proposals for GPU accelerated simulators. These include simulations using a large number of qubits and memory [11], using proprietary frameworks such as CUDA [3][10].

To the author’s knowledge, QCGPU is the first open source quantum computer simulator to use the functionality provided by OpenCL. The advantages/disadvantages of which (over CUDA or similar frameworks) are discussed in section 2.2.

2.2 OpenCL

OpenCL (Open Computing Language) is a general-purpose framework for heterogeneous parallel computing on cross-vendor hardware, such as CPUs, GPUs, DSP (digital signal processors) and FPGAs (field-programmable gate arrays). It provides an abstraction for low-level hardware routing and a consistent memory and execution model for dealing with massively-parallel code execution. This allows the framework to scale from embedded systems to hardware from NVidia, API, AMD, Intel and other manufacturers, all without having to rewrite the source code for various backends. An overview of OpenCL is given in [20].

The main advantage of using OpenCL over a hardware specific framework is that of a portability first approach. OpenCL has the largest hardware coverage, and as a library only it requires no tool dependencies. Aside from this, OpenCL is very well suited to tasks that can be expressed as a program working in parallel over simple data structures (such as arrays/vectors). The disadvantages with OpenCL, however, come from this lack of a hardware-
specific approach. Using proprietary frameworks can sometimes be faster than using OpenCL [13], and sometimes it can also be more straightforward to develop kernels for the devices.

OpenCL is an open standard maintained by the non-profit Khronos Group. It views a computing system as a number of compute devices (such as CPUs or accelerators such as GPUs), attached to a host processor (a CPU). OpenCL executes functions on these devices called Kernels, and these kernels are written in a C-like language, OpenCL C. A compute device is made up of several compute units which contain multiple processing elements. It is the processing elements that execute kernels. This is shown in figure 1.

At the host level, a compute device is selected. The OpenCL API then uses its platform later to submit work to the device and manage things like the work distribution and memory. The work is defined using kernels. These kernels are written using a language called OpenCL C, that executes in parallel over a predefined, n-dimensional computation domain. Each independent element of this execution is a work item. These are equivalent to NVIDIA CUDA threads. The groups of work items, work groups, are equivalent to CUDA thread blocks.

With this, a general pipeline for most GPGPU OpenCL applications can be described. First, a CPU host defines an n-dimensional computation domain over some region of DRAM memory. Every index of this n-dimensional domain will be a work item, and each work item will execute the same given Kernel.

The host then defines a grouping of these into work groups. Each work item in the work-groups will execute concurrently within a compute unit and will share some local memory. These are placed on a work queue.

The hardware will then load DRAM into the global GPU RAM, and execute each work group on the work-queue.

On the GPU, the multiprocessor will execute 32 threads at once. If the work group contains more threads, they will be serialized.

There are some limitations. The global work size must be a multiple of the work group size. This is to say the work group must fit evenly into the data structure.

Secondly, the number of elements in the n-dimensional vector must be less or equal the CL_KERNEL_WORK_GROUP_SIZE flag. This is important to the QCGPU library as it sets a hard limitation on the size of the state vector being stored on the GPU. CL_KERNEL_WORK_GROUP_SIZE is a hardware flag, and OpenCL will return an error code if either of these conditions is violated.

3 Implementation

The implementation of a quantum computer simulator can be broken down into three main parts, the representation of the state/quantum register, the representation and application of gates and measurement.

3.1 Quantum Computing

In classical computation, a bit can be either 0 or 1. This can be used to represent numbers by using them together and represent them in binary. With n bits, $2^n$ numbers can be represented. For example, if $n = 2$, 4 numbers can be represented, 0 = 00, 1 = 01, 2 = 10 and 3 = 11. Multiple bits, however, can only ever represent one number at a time.

A quantum computer is a computer that uses the phenomena from quantum mechanics (such as superposition, collapse, uncertainty and entanglement) to perform computations. In quantum computation, the analogue of a bit is a qubit, a quantum bit.

A qubit is a quantum system in which the Boolean states 0 and 1 are represented by a specified pair of normalized and mutually orthogonal (orthonormal) quantum states, labelled $\{|0\rangle$ and $|1\rangle\}$. These states form a computational basis and are called basis states. Any other state of the qubit, if it is not entangled (pure), can be written as a sum $\alpha_0|0\rangle + \alpha_1|1\rangle$. This sum is called a superposition. The coefficients, $\alpha$ and $\beta$ are constrained such that $|\alpha|^2 + |\beta|^2 = 1$ (the normalization condition), and $\alpha, \beta \in \mathbb{C}$. Qubits can be any quantum system, but are typically an atom, a nuclear spin or a polarized photon.

A collection of n qubits is called a quantum register of size n. Similar to that of classical computing, a quantum register can be used to represent information in binary form. For example, a number 6 can be written as a register in the state $|1\rangle \otimes |1\rangle \otimes |0\rangle$. There
is alternate notation for the same state, $|110\rangle$ or $|6\rangle$. These can usually be distinguished through context.

As with collections of classical bits, $n$ qubits can represent a number up to $2^n-1$, including 0. However, using qubits, you can store two states simultaneously. For example, instead of preparing one of the basis states, we can prepare the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Adding two extra qubits we can prepare the states 3 and 7, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |1\rangle \otimes |0\rangle \equiv \frac{1}{\sqrt{2}}(|3\rangle + |7\rangle)$. Thus, the state of a register can take any value corresponding to a linear combination of the basis states, given that the coefficients meet the normalization constraint.

The state of a qubit or multiple qubits can be written as a vector. The most common basis and the one used in both this paper and the software is

$$\{|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\}$$

It should be noted here that the notation $|$ is Dirac notation, a standard notation system used for the representation of quantum states [7].

The standard notation of an arbitrary state is

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{2^n-1} |1\ldots1\rangle$$

$$= [\alpha_0 \ \alpha_1 \ \cdots \ \alpha_{2^n-1}]^T.$$

It should also be noted that $\otimes$ represents the tensor product, and corresponds to the kronecker product of vectors and matrices.

There are $2^n$ elements in the vector as there are $2^n$ possible states which can be represented. Recall that $\alpha \in \mathbb{C}$, thus to store the state of $n$ qubits, you must store $2^n$ complex numbers. It is this that is the main challenge presented in the simulation of quantum computers: the exponential growth of the state vector.

It is impossible to experimentally find the complete state of an arbitrary quantum system. When measuring a qubit, the result of the measurement, whether 0 or 1, depends on the coefficient of that state. Given a state $|\psi\rangle$ (as above), the probability of getting an outcome $j$ from a measurement is

$$P(j) = |\alpha_j|^2$$

It should now also be clear why the normalization constraint exists, as there must be a probability of 1 that some outcome will happen when doing a measurement.

For example, the state $|0\rangle$ has the state vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}^T$, thus the probability of measuring 1 is $|\alpha_0|^2 = |1|^2 = 1$, so measuring the state $|0\rangle$ will always result in the outcome 0.

This is not the same however with the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The probability of getting a 0 when this state is measured is $|\alpha_0|^2 = \frac{1}{\sqrt{2}} = 0.5$, so there is a 50% chance that 0 will be measured, which implies that there is also a 50% chance that 1 will be measured. This is the nature of superposition.

As quantum registers are represented as vectors, their manipulations (in the form of quantum gates) are represented as matrices. These matrices have a number of requirements

Just as we have defined the state of qubits using vectors, we can also describe how the state changes over time with linear algebra.

In quantum computing, the state of a qubit is changed by using a quantum logic gate, or just quantum gate. These gates are similar to classical logic gates. A quantum gate operates on a quantum register to evolve its state. Mathematically, gates are represented as matrices. To represent a quantum gate, a matrix must conform with the postulates of quantum mechanics as they multiply a state vector to produce an evolved state.

A matrix must preserve probability (preserve the normalization). Matrices that ensure this property are called unitary. Formally, a matrix $U$ is unitary if it satisfies the property that its conjugate transpose $U^\dagger$ is also its inverse, $U^\dagger U = UU^\dagger = I$, where $I$ is the identity matrix.

In quantum computing, all gates have a corresponding unitary matrix, and all unitary matrices have a corresponding quantum gate.

The physical realization of a gate depends on the realization of the qubit. For example, if a qubit is represented by ions in an ion trap, a tuned laser can induce a unitary transformation, acting as a quantum gate.

Gates that act on a single qubit (represented by a $2 \times 2$ matrix) can be applied to a quantum register with an arbitrary number of qubits. For example, if we want a gate $U$ to act on the third qubit of a four-qubit register, the full gate is formed by $I \otimes I \otimes X \otimes I$.

In general, a gate $U_t$, acting on a $n$ qubit register, only affecting the $t$th qubit, is formed by

$$U_t = \bigotimes_{j=1}^{n} \begin{cases} U & j = t \\ I & \text{otherwise} \end{cases}$$

Multiple gates can be applied to a register. There are a number of common gates used in Quantum Computing, such as the Hadamard gate ($H$), the Pauli Gates ($X, Y, Z$) and the phase shift gate among others.

Gates can also be controlled, or only applied in the case of a specified qubit being 1. An example of this type of gate is the $CNOT$ gate.
3.2 Implementing Quantum Registers

The most straightforward way to implement quantum registers using OpenCL is with a section of memory on the accelerator. This is equivalent to storing the whole state vector in memory. The alternative to this is using some form of a sparse vector, where only the nonzero elements are stored in memory. The problem with this sparse vector approach is that it requires the overhead of storing the location of all elements, for which the memory (and additional lookups) can reduce the speed and overall memory efficiency, especially for complex or highly entangled states. The downside to not using a sparse vector is that the creation of registers can be slower, and when only doing operations on a small number of qubits (where the other qubits are not altered), the performance difference is clear. (see figure 8, which compares QCGPU to Libquantum, which uses sparse state vectors).

In QCGPU, when a quantum register is initialized with \( n \) qubits, a section of memory is used, set to \( 0 + 0i \). Then, a single element is set to \( 1 + 0i \), as the register must be normalized. A reference to this section of memory, the OpenCL program, queue and context, and the number of qubits are used to represent the state of the quantum register.

4 Implementing Quantum Gates

As previously states, gates are represented mathematically as matrices. The na"ive approach to implement quantum gates is to represent these gates. The problem exists in that they are \( 2^n \times 2^n \) square matrices, and thus require \( 2^{2n} \) complex numbers to represent them. This coupled with their generation matrices, and thus require \( 2^n \times 2^n \) complex numbers to be represented, compared to \( 2^{2n} \).

There is an alternative approach, which can be done in \( O(n) \) time. By only using single qubit gates, a gate can be applied without having to store the state.

4.1 Implementing Measurement

The measurement process relies on knowing the probability of each output state. The actual selection of an outcome based on these probabilities cannot be parallelized, however, the calculation of the probabilities can be.

From this, an outcome can be selected. Because the probabilities can be calculated separately to the measurement, it also allows multiple measurements to be made. While this isn’t possible on a quantum computer, it does mean that it is easier to prototype/simulate algorithms, the primary goal of the software library.

5 Benchmarking Data

In this section, benchmarks of the QCGPU library are presented. These are for the purpose of showing the performance of the simulation library, using hardware close to what would typically be used. Also included are comparisons against similar libraries, which are detailed in the various subsections.
All of the following benchmarks were run on an AWS EC2 P3.2xLarge instance with a 25GB general purpose SSD, an NVidia Tesla V100 16GB GPU, 8 vCPUs and 61GB RAM.

5.1 Core Benchmarks

Here benchmarks of the main features of QCGPU are presented. Benchmarked in figures 2, 3, 4, 5 and 6 are various functions, run with 5 and 25 qubits.

For the simulation of a $n$ qubit system, the simulator requires approximately $8 \cdot 2^n$ bytes of memory, actual results can differ, being variant based on the methods being used among other factors.

5.2 QISKit

QISKit [1] is a python framework for interacting with the IBM quantum experience. It includes a simulator, written in python. The included simulator is based on using a NumPy vector to represent the state of the quantum computer, and matrices to represent the gates. Here, QCGPU is benchmarked against it.

The benchmarks are done by creating a $n$ qubit register, applying a Hadamard gate to all of them and measuring 1000 times. The results are shown in figure 7.

5.3 Libquantum

Libquantum [5] is a C library for the simulation of quantum algorithms. It is quite similar in function to QCGPU, but it is based all on the CPU and uses sparse state vectors. The use of sparse state vectors accounts for the speedup over QCGPU for things such as register initialization. Figure 8 shows a comparison of benchmarks between libquantum and QCGPU, using 25 qubits.

6 Conclusion

The previous chapters have explored the implementation of a library for the simulation of quantum computers, using hardware acceleration through the OpenCL framework. Although time-consuming for a large number of qubits (relative to hardware), the simulation of quantum computers is a necessary part
The field of quantum computation is rapidly expanding, and with it, the areas of research are growing too. There is clearly a lot of room for expansion in the field of quantum algorithms, where the usefulness of various algorithms will be improved over time.

There is also room to improve in the simulator. There are some algebraic manipulations which can be done when the whole circuit is known beforehand, which are usually done using a general-purpose quantum computing language or framework [18] [8] [15].

Currently on the roadmap for the development of this library is enabling the library for use on distributed systems, along with implementing additional algorithms using the simulation library. Also planned is the integration of some proprietary kernels (using a framework such as CUDA) which can offset some of the downsides of using OpenCL.
References

[1] Qiskit/qiskit-sdk-py. URL https://github.com/QISKit/qiskit-sdk-py.
[2] Quantiki, list of qc simulators. https://www.quantiki.org/wiki/list- qc-simulators.
[3] Andrei Amariutei and Simona Caraiman. Parallel quantum computer simulation on the gpu. In System Theory, Control, and Computing (ICSTCC), 2011 15th International Conference on, pages 1–6. IEEE, 2011.
[4] Daniel J Bernstein, Nadia Heninger, Paul Lou, and Luke Valenta. Post-quantum rsa. In International Workshop on Post-Quantum Cryptography, pages 311–329. Springer, 2017.
[5] B Butscher and H Weimer. libquantum. the c library for quantum computing and quantum simulation. 2004–2013.
[6] Simon J Devitt. Performing quantum computing experiments in the cloud. Physical Review A, 94(3): 032329, 2016.
[7] P. A. M. Dirac. A new notation for quantum mechanics. Mathematical Proceedings of the Cambridge Philosophical Society, 35(3):416–418, 1939. DOI: 10.1017/S0305004100021162.
[8] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In ACM SIGPLAN Notices, volume 48, pages 333–342. ACM, 2013.
[9] Lov K Grover. A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 212–219. ACM, 1996.
[10] 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. ISSN 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2009.09.021. URL http://www.sciencedirect.com/science/article/pii/S0010465509003117.
[11] Thomas Hänner 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.
[12] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature, 549(7671):242, 2017.
[13] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. arXiv preprint arXiv:1005.2581, 2010.
[14] Nikolaj Moll, Panagiotis Barkoutsos, Lev S Bishop, Jerry M Chow, Andrew Cross, Daniel J Egger, Stefan Filipp, Andreas Fuhrer, Jay M Gambetta, Marc Ganzhorn, et al. Quantum optimization using variational algorithms on near-term quantum devices. arXiv preprint arXiv:1710.01022, 2017.
[15] Bernhard Ömer. Qcl-a programming language for quantum computers. Software available on-line at http://tph. tuwien. ac. at/~oemer/qcl. html, 2003.
[16] Diego Ristè, Marcus P Da Silva, Colm A Ryan, Andrew W Cross, Antonio D Córcoles, John A Smolin, Jay M Gambetta, Jerry M Chow, and Blake R Johnson. Demonstration of quantum advantage in machine learning. npj Quantum Information, 3(1):16, 2017.
[17] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review, 41(2):303–332, 1999.
[18] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. arXiv preprint arXiv:1608.03355, 2016.
[19] Jordan Stephen. Quantum algorithm zoo. 2018.
[20] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. Person Education, 49, 2012.
[21] Chuan Wang, Fu-Guo Deng, Yan-Song Li, Xiao-Shu Liu, and Gui Lu Long. Quantum secure direct communication with high-dimension quantum superdense coding. Physical Review A, 71(4):044305, 2005.
[22] Christof Zalka. Grover’s quantum searching algorithm is optimal. Physical Review A, 60(4):2746, 1999.