Quantum Computer Simulation Using CUDA

Alexander Smith Khashayar Khavari University of toronto
Department of Electrical and Computer Engineering

I. Introduction

Quantum computing has captured the attention of many researchers in the past decade. While researchers in the fields of electrical engineering and physics have concentrated on realizing a physical machine that satisfies the criteria of a quantum computing, others in the research community are developing algorithms that can take advantage of such machines. Through exploitation of the inherit superposition observed at the quantum level, many interesting quantum algorithms (q-algorithms) have been developed. These range from simple communication between two points using super-dense coding to factorization of a large number into its prime components through Shor's algorithm.

Despite the dedication and hard work of physicists and engineers, a physically stable quantum computer is still a dream to be fulfilled. To deal with this problem and to encourage development of more q-algorithms, it has been proposed to simulate the functionality of a quantum computer using a classical one. Quantum computer simulation allows researchers to validate existing quantum algorithms without the need for a physical quantum computer. However, the inherent complexity of a quantum system results in extremely time-consuming simulations on a classical machine.

In a physical quantum computer, quantum superposition of states allows the simultaneous manipulation of all possible combinations of a set of bits in a single operation, speeding up many algorithms exponentially when compared to a classical computer. This is main challenge that a classical simulator has to face. Because of this, quantum algorithms, which are supposed to reduce exponential processing times to linear ones, run more slowly than their classical counterparts when simulated. However, the inherit parallelism involved in simulating a quantum system makes it suitable for GPU implementations.

In this project we will implement a simulator for Quantum Fourier Transform (QFT) using CUDA. Like classical Fourier transform, the QFT is at the heart of many other algorithms. Probably the most famous example is Shor's integer factorization algorithm [3] which is a set of protocols that convert the factorization problem into a period detection problem.

In order to validate our implementation and to test it's performance we will use *libquantum* as the basis for this work. *libquantum* is a set of a programming libraries written in C for the simulation of quantum computers. There are many quantum algorithms implemented by this library, and its gate-by-gate approach to simulation allows a fair comparison and easy integration of our implementation.

The remainder of this paper is organized as follows. We start by introducing the reader to the basics of Quantum Fourier Transform in Section II. We then present our approach for a CUDA implementation in Section III. In Section IV we present methods of improving the performance of our code for GPUs. Section V presents a subset of our experiments to evaluate the correctness, accuracy and performance of our design. We summarize the work and point to future improvements in Section VI.

II. Quantum Fourier Transform

Fig. 1 presents the quantum circuit for QFT, while Equation 1 describes the operation of the overall system in terms of the input and output states [2]. Each input state goes through a number of phase shift gates and Hadamard transforms. However, the phase shift gates "controlled" quantum gates. This means that the effect the have on the target qubit (quantum bit) is dependent on the value of the control qubit. A gate-by gate simulation of this circuit involves evaluating the effect of each gate on each qubit for all different states. Note that given the superposition property of quantum computation, n input qubits translate to $N = 2^n$ states. What makes this problem suitable for a CUDA implementation is that processing each state is completely independent from the other $N - 1$ states.

$$|o\rangle = |i_1, \ldots, i_n\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi j \frac{i.k}{N}} |k\rangle \tag{1}$$

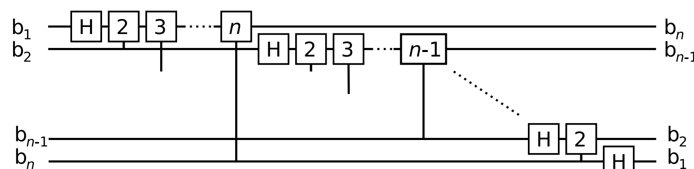


Fig. 1. Quantum Fourier Transform circuit consisting of controlled phase shift gates and Hadamard transforms.

Before we start describing our approach to solving this problem, we would like to clarify the operation of the QFT through an example with two qubits ($n = 2$). The number of states for this example is $N = 2^2 = 4$ and the states are $|0>$, $|1>$, $|2>$, $|3>$. Each state is represented as a column vector of 4 elements with a single nonzero element at the row corresponding to the state number, eg. $|2>= [0010]^T$. The input state to our system is a weighted sum of these four states, where the coefficients are complex numbers, Equation 2.

$$|\Psi_{in}>= \sum_{k=0}^{N-1} \alpha_k \, |k> \tag{2}$$

The output state is also a weighted sum of the same basis states with different coefficients, Equation 3.

$$|\Psi_{out}>= \sum_{k=0}^{N-1} \beta_k \, |k> \tag{3}$$

In order to describe the output of the circuit in terms of its input, we need to describe the output coefficients β_k as a function of input coefficients α_k. It is a well-known fact that the operation of each gate on any qubit q can be described as a linear, unitary transformation on its input qubits. As it turns out, the output coefficient β_i is a function of α_i and $\alpha_{i \oplus 2^q}$, where q is the index of the qubit at which that the gate is being applied. We note that the same two input coefficients are required to obtain both β_i and $\beta_{i \oplus 2^q}$.

Applying what we have reviewed to our example for state $|2>$ and qubit $q = 1$ we have the following transformation after the Hadamard gate.

$$\beta_2 = \frac{1}{\sqrt{2}} \left(\alpha_2 - \alpha_0\right) \tag{4}$$

The effect of all other gates on all states can be evaluated in a similar manner.

III. Implementation in CUDA

We considered two different approaches to the problem. An *emulator*, in which each gate is implemented in CUDA and the whole circuit is implemented by interconnecting these gates, and a *simulator*. A simulator achieves the the same final result, but through a different method. In our case, our emulator uses matrix multiplication.

A. Simulator

One can show that any quantum circuit can be reduced to an algebraic problem involving the multiplication of a matrix representing all of the gates in the circuit in an $N = 2^n$ dimensional space by the input state vector. This is due to what is known as “delayed measurement” in quantum systems. Simply stated, this property allows one to carry out operations that depend on the intermediate value of qubits without actually measuring the values of those bits except for a single measurement at the end of the circuit. For example, the following matrix can be used to evaluate the QFT of a three-bit input circuit.

$$M = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & 1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w \end{bmatrix}$$

Using this idea, we reduced the QFT circuit to a problem of multiplying an $N \times N$ matrix with all input states and then summing the results; or, equivalently:

$$|\Psi_{out}>= \sum_{k=0}^{N-1} \alpha_k M \times \, |k>= M \times \, |\Psi_{in}> \tag{5}$$

We implemented both a CPU and an optimized GPU version of this approach, achieving three orders of magnitude in speedup. However, it soon became clear that although this approach is very GPU-friendly, it has a complexity of $O(N \times N) = O(2^{2n})$. As we will see in the next section, a gate-by-gate emulation has a complexity of $O(n^2 \times 2^n)$. For this reason, the overall speed of the gate-by-gate emulation is much higher for large values of n. Because this approach ultimately was not fruitfal, and to save space, we will not present details of our optimizations for this matrix-multiplication method here.

q		0	1	...	7	8
	I	I xor 2^q	I xor 2^q	...	I xor 2^q	I xor 2^q
Block 1	0	1	2	...	128	256
	1	0	3	...	129	257
	2	3	0	...	130	258
	3	2	1	...	131	259
	4	5	6	...	132	260
	...	...	...	...	...	...
	511	510	509	...	383	255
Block 2	512	513	514	...	640	768
	513	512	515	...	641	769
	514	515	512	...	642	770
	515	514	513	...	643	771
	516	517	518	...	644	772
	...	...	...	...	...	...
	1023	1022	1021	...	895	767

Fig. 2. Memory access pattern for the last nine qubits.

B. Emulator

Our QFT emulator simulates the quantum circuit gate by gate. The QFT implementation in *libquantum* is a direct implementation of a modified form of the circuit shown in Fig. 1. The authors of *libquantum* have applied an algebraic manipulation of the circuit so qubits are applied to the high-order gates first, which is opposite to what is shown in Fig. 1. This manipulation does not affect the result, nor the number of computations required, and provides no additional opportunity for optimization. Nevertheless, we chose to follow *libquantum*'s form of the circuit in order to facilitate testing of our code.

Our initial emulation algorithm is a fairly direct port of the *libquantum* code to CUDA. Performing a QFT (using *libquantum*'s method) requires applying several controlled quantum gates to the last output qubit, then applying several gates to the second-last qubit, and so on. From the discussion in Section II, it is clear that several consecutive gates applied to the same qubit will involve the same state coefficients. Therefore, we can combine each set of consecutive gates on a qubit (several phase-shifts and a Hadamard transformation) into a single kernel call. This results in n kernel calls for an n-qubit QFT. Also from Section II, we see that state calculations can be grouped: calculating the outputs of states i and $i \oplus 2^q$ both *require* and *produce* coefficients α_i and $\alpha_{i\oplus 2^q}$. Thus in CUDA, we can assign one thread to every pair of coefficients.

IV. Optimization for GPU

The largest qubit state vector On the lab machines, the largest qubit state vector that fits in GPU memory is 26-bits (requiring 512 MB of memory). Our initial GPU implementation gave a $33\times$ speedup over *libquantum*. From here, we applied several different types of optimizations. We used some methods from [1], as well as others described below.

A. Algebraic Manipulations

Trignometric floating-point operations are some of the slowest operations on CUDA GPUs. We began our optimizations by trying to reduce the number and complexity of floating point operations required. Our first optimization involved combining consecutive phase shift gates into one. *libquantum* simulates each gate separately. For n consecutive phase shift gates, this approach requires n $\sin$ and $\cos$ calculations, n complex multiplications (each requiring $4n$ floating-point multiplications and $2n$ additions), and $2n$ further floating-point additions. Since quantum phase shift gates commute, we can group the phase shift gates together. We first calculate the total resulting phase shift (which will be different for each state), and then perform a single floating point $\sin$ and $\cos$. This procedure alone gave an additional $3.8\times$ speedup, for a total so far of $125\times$. See Section V for details.

On the CUDA architecture, `sinf` and `cosf` are much slower than `sqrtf`. Since $\sin\phi = \pm\sqrt{1-\cos^2\phi}$, we can achieve a further speedup by replacing a call to `sinf` with `sqrt`. Finally, we can halve the number of accesses to global memory by combining the phase shift and Hadamard calculations into a single function. Instead of fetching and writing back a pair of state coefficients for the phase, and then the Hadamard gate, the coefficients are fetched into thread-local variables once at the start of the function. The phase shift is applied, followed by the Hadamard transformation, and only then is global memory accessed again in order to write the final result back.

B. Combining Kernels and Shared Memory

One of the main bottlenecks in the CUDA architecture is the memory access speed. Accessing global memory takes several hundred clock cycles. If memory accesses are not coalesced, the time penalty for accessing memory is compounded. In our algorithm, there is a separate kernel call for each qubit $q = n-1\ldots 0$. Some particular thread will be performing calculations on coefficients α_i and $\alpha_{i\oplus 2^q}$. An important question is: If thread i accesses coefficient α_i and then $\alpha_{i\oplus 2^q}$, will these memory accesses coalesce? The answer can be found in Fig. 2. Accesses to α_i obviously will coalesce, since sequential threads are accessing sequential memory locations. Accesses to $\alpha_{i\oplus 2^q}$ are uncoalesced for low values of q, but do coalesced when q is

q		9	10	11
	I	I xor 2^q	I xor 2^q	I xor 2^q
Block 1	0	512	1024	2048
	...	...	...	...
	511	1023	1535	2559
Block 2	512	0	1536	2560
	...	...	...	...
	1023	511	2047	3071
Block 3	1024	1536	0	3072
	...	...	...	...
	1535	2047	511	3583
Block 4	1536	512	512	3584
	...	...	...	...
	2047	1023	1023	4096
Block 5	2048	2560	3072	0
	...	...	...	...
	2559	3071	3583	511
Block 6	2560	2048	3584	512
	...	...	...	...
	3071	2559	4096	1023
Block 7	3072	3584	2048	1024
	...	...	...	...
	3583	4096	2559	1535
Block 8	3584	3072	2560	1536
	...	...	...	...
	4096	3583	3071	2047

Fig. 3. Memory access pattern for qubits 9–11.

high enough. Therefore, we should be able to achieve a speedup by improving global memory access for the least significant few bits.

Fig. 2 illustrates another useful features of the QFT algorithm. For qubits $0 \ldots 8$, the pairs of coefficients α_i and $\alpha_{i \oplus 2^q}$ lie within the same 512-element block of the vector state array. 512 complex numbers is the largest power of two we can fit into shared memory on the GTX280 graphics cards. This observation suggests a means of using shared memory to improve performance. Each block will copy-in 512 elements from the state vector. The block will then perform the calculations for the last nine qubits[1] in its own shared memory and then copy the results back. Copying coefficients into and out of shared memory can be done with global memory coalescing. In addition, this reduced the overhead incurred in launching nine separate kernels. While implementing this optimization, we were careful to avoid shared memory bank conflicts as much as possible.

Our use of shared memory in the last nine qubits is possible because the coefficients for those qubits naturally fall into disjoint groups of less than 512 $(= 2^9)$. For higher qubits, the coefficient pairs spread out in memory. However, for any r consecutive qubits, the state vector coefficients can be divided into disjoint sets such that the calculations for the coefficients in that set affect those coefficients, and those coefficients only. In other words for any r consecutive qubits, the coefficients can be divided into *disjoint* sets. Several of these sets can be copied into shared memory, and a block can process them completely without worrying about synchronization with any other blocks. Fig. 3 illustrates how a disjoint set can be found for qubits 9 to 11 ($r = 3$). In this case, state coefficients 3584 and 3072 are paired, since $3584 \oplus 2^9 = 3072$. These coefficients are highlighted in yellow in the figure. In order to apply the gates for qubit 9, we must first have the output coefficients from qubit 10. For qubit 10, these two coefficients are paired with another two coefficients: 2048 and 2560. These four coefficients in turn require four more coefficients from qubit 11. Thus, for qubits 9–11, the coefficients $\{0, 512, 1024, 1536, 2048, 2560, 3072, 3584\}$ form a closed set. The gates for qubits 11 through 9 can be applied to these coefficients without regard to any other coefficients in the state vector. We attempted to implement an algorithm that copied several such closed sets of coefficients into shared memory, processed them there, and then wrote them back. However, we ran into difficulties with the implementation and were unable to get the algorithm working in time for the project demo.

V. Evaluation

We have evaluated the correctness of our implementation and all improvements by comparing the results generated by the GPU to those generated by *libquantum*. We compared our calculated output vector with that from *libquantum* by calculating the l_2-norm between them. The original GPU implementation before optimizations agreed exactly with the CPU version. The algebraic manipulations introduced in Section IV-B introduce a small error. In our tests, the error has never exceeded 5×10^{-5}, and given that the outputs represent the probability of a specific state occurring in the quantum system, this small error is

[1]Recall from Section III-B that we process qubits in reverse order.

bits	10	12	14	16	18	20	22	24	26
CPU [ms]	0	1.915	10.51	40.42	247.0	1342	6272	29163	134627
GPU [ms]	0.23	0.23	0.66	2.00	8.65	40.18	188.16	876.06	4096.00
Speedup	0	6.4	15.9	20.2	28.6	33.4	33.3	33.3	32.9

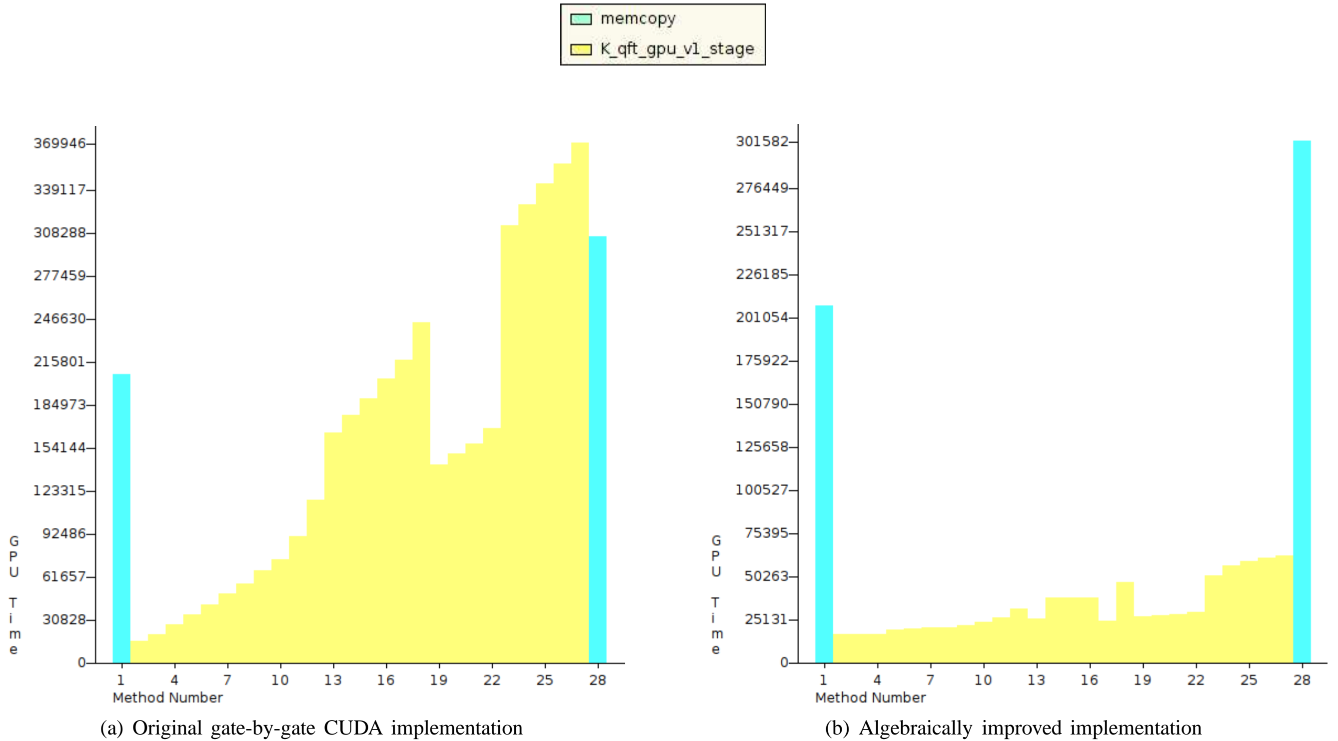
TABLE I

SPEEDUP ACHIEVED BY PARALLELIZING THE PROCESS OF ALL STATES FOR EACH BIT VALUE INTO A SINGLE KERNEL CALL.

negligible. In fact *libquantum* continually discards states with probabilities on this order, intentionally rounding them down to zero.[2]

We also compared the performance of our implementations against *libquantum*. Table V presents the speedup gained by our first CUDA implementation. Using n CUDA kernel calls, each of which process all $N = 2^n$ states for a specific qubit, we have gained a factor of about 33 times in speedup.

In order to demonstrate the advantage of each subsequent optimization phase, we have presented the output from the CUDA Visual Profiler. Fig. 4 presents the processing time samples of the profiler for the original and the algebraically optimized versions of the code. The optimizations include the reduction of phase calculations, the elimination of the $\sin$ function, the reduction of global memory access by half, and the combination of phase shift gates and a Hadamard gate into a single, more complex gate.



(a) Original gate-by-gate CUDA implementation

(b) Algebraically improved implementation

Fig. 4. Combining gates and reducing computation complexity of each thread results in much smaller processing time.

Fig. 5 demonstrates how combining the last nine kernel calls and using shared memory results in shorter processing times. Note that the green bar represents the time it takes to achieve the same work done by the last nine yellow bars from Fig 4(a).

Finally we compare the speedup achieved through each stage of optimization in Fig. 6. We have included the results from combining kernels from higher bits when using global memory. This implementation performs worse than keeping each kernel separate due to the amount of uncoalesced memory accesses. Since we have not been able to resolve the problem we are experiencing with the case of using shared memory for higher bits, we have not presented the performance results here. Table V presents the overall speedup and incremental speedup achieved for each of the optimization steps.

VI. CONCLUSION

Quantum computer simulation, although extremely time consuming, is currently a necessary part o developing and testing new quantum algorithms. Through a CUDA implementation of Quantum Fourier Transform, a commonly used algorithm, we

[2]The fact that *libquantum* does this also makes it impossible to draw any further conclusions about accuracy. Small discrepancies in our algorithm could be caused because we *do not* drop any states until the final comparison step.

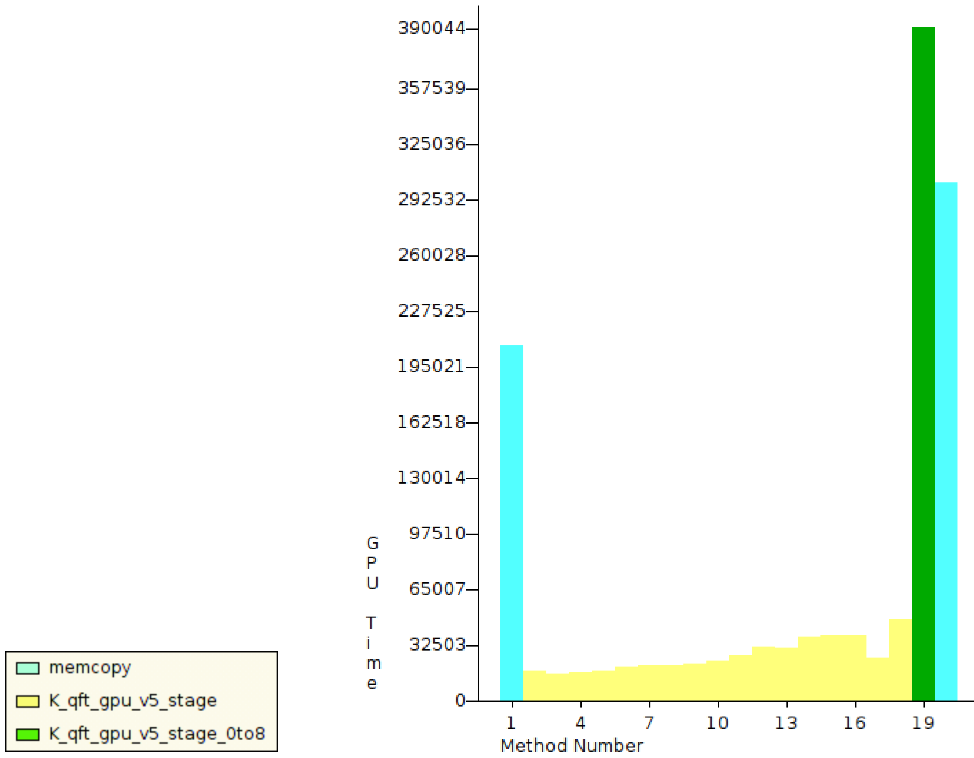


Fig. 5. Combining the last nine kernel calls into a single kernel that uses shared memory.

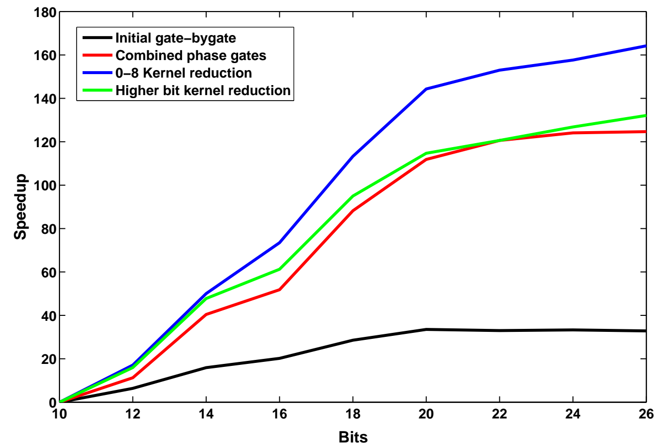


Fig. 6. Speedup comparison of different optimizations.

Algorithm	Improvement	Time [ms]	Overall Speedup	Incremental Speedup
libquantum	—	134627	1	1
1	Plain	4094.77	32.88	32.88
2	Phase Gates	1076.76	125.03	3.8
3	Trigonometry	982.4	137.04	1.1
4	Shared Mem	837.85	160.68	1.17
5	0To8 Kernel	828.2	162.55	1.01
6	Combined Kernels	1020.96	131.86	0.81

TABLE II

OVERALL AND INCREMENTAL SPEEDUP FOR EACH OPTIMIZATION STEP.

have shown how GPUs can help speed up quantum computer simulations. After a number of optimizations ranging from algebraic manipulations to reduce computation complexity to use of combined kernels, shared memory and reduced bank conflicts, we have achieved speed ups of over 160 times. Unfortunately, given the limited time, we have not been able to find the error in the logic of our final optimization plan. Given our current experiments in this direction, we would expect to see a further speedup up to a factor of two times faster.

References

[1] E. Gutierrez, S. Romero, M. A. Trenas, and E. L. Zapata, *Computational Science — ICCS 2008*. Springer Berlin, 2008, vol. 5101, ch. Parallel Quantum Computer Simulation on the CUDA Architecture, pp. 700–709.

[2] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[3] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *35th Annual Symposium on Foundations of Computer Science*, Nov 1994, pp. 124–134.