Accelerator Circuits for Quantum Simulation

Yuranan Kitrungrotsakul, Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Phayathai Rd., Pathumwan, Bangkok, Thailand, 10330
yuranan.kit@student.chula.ac.th, prabhas.c@chula.ac.th

***Abstract*— Although a quantum computer is the future of computing, its practical implementation is still far off. Programming a quantum computer is also difficult. Thus, using a quantum computer simulator is a way to learn how to use a quantum computer. QCL is one of the quantum computer simulators. It can simulate the quantum environment and execute quantum computer programs. However, the quantum computer simulator has limited storage due to its data structure that simulates quantum bits. It takes long time to simulate a large number of quantum bits. This work proposes an accelerator of quantum simulator which is implemented in hardware circuits with Field Programmable Gate Array technology.**

Keywords—quantum computer; QCL simulator; acceleator circuits;

I. INTRODUCTION

In every year, computers are developed to be smaller, execute tasks faster and use lower power due to the technology. The fabrication technology is able to achieve double circuit density every year continuously for almost 40 years. Gordon Moore has forecasted that and it is accepted as Moore's law.

High performance computers are required to solve many problems that cannot be solved with current generation of computers such as in medicine, in science and so on. One of the promising types of high performance computer is based on using Quantum effect for computation. In order to create a quantum computer, the fundamental storage, quantum bit, that holds simultaneous many states must be realized. The quantum computer is being created in research laboratories with few quantum bits. Only one system has been available commercially. D-wave systems [1] announced the first commercial quantum computer operating on a 128-qubit in 2011. However, it does not have any of evidence, which prove that it operates with the real quantum effects. In order to study the behavior of a quantum computer, many of the quantum simulators were created. The quantum simulator simulates the behavior of quantum computer on classical computers. Users can write programs for quantum computers that are executed by classical computers via simulators.

In order to simulate operations of quantum bits, the simulator must calculate all states that are in entanglement. This calculation consumes both time and space. Due to this constraint, the simulator can only work on the small number of quantum bits. This work tackles the aspect of speed up the simulator. The work proposed an accelerator circuit for a particular simulator, QCL (Quantum computer language). This simulator has been widely available. It is stable and open source.

The next section describes the basic of quantum computing. Section 3 describes the concept of QCL simulator. Section 4 and 5 describe how to approach the problem of finding where in the simulator to be replaced by a hardware circuit. Section 6 gives the details of the design of the accelerator circuit. Section 7 shows the experimental result and the conclusion is in Section 8.

II. ESSENTIAL QUANTUM COMPUTATION

A. Quantum bit

A quantum computer [2] is totally different from the classical computer. Quantum computer's operations are based on the theory of quantum mechanics. The smallest unit of quantum computers is a quantum bit (Qubit). The qubit represents a quantum particle, which has a superposition property. Thus, the qubit is represented in a linear combination as shown in equation 2.1.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (2.1)$$

The α and β are complex numbers. Due to the fact that the qubit carries value in superposition state, it has the value 0 and 1 in the same time. This is a major fact that differentiates the quantum computer from the classical computer which has only one value either 0 or 1 in each bit. The qubit has value 0 with probability α^2 and value 1 with probability β^2. In order to preserve the law of total probabilty, the value of α and β must satisfy equation 2.2.

$$|\alpha|^2 + |\beta|^2 = 1 \quad (2.2)$$

Due to the law of total probability, equation 2.1 can be rewritten with equation 2.2. The new equation is equation 2.3.

$$|\psi\rangle = e^{i\gamma}(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle) \quad (2.3)$$

978-1-4799-1966-6/15/$31.00 ©2015 IEEE

The γ, θ and φ are real numbers and the $e^{i\gamma}$ can be ignored because it does not affect the value of qubit. Thus, equation 2.3 can be rewritten into equation 2.4.

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \qquad (2.4)$$

Equation 2.4 is explained by Bloch sphere in figure 1.

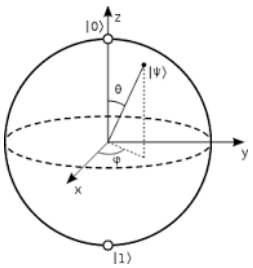

Figure.1: A qubit representation in Bloch sphere

From the figure 1, the $|\psi\rangle$, which is the value of qubit, is pointed to somewhere on the surface of the Bloch sphere depended on the value of φ and θ, which are an angle with the X and Z axis of Bloch sphere. Thus, if $|\psi\rangle = |0\rangle$ means it points to the highest of Bloch sphere, vice versa for one.

In case of multiple qubits, equation 2.1 changes in to equation 2.5 due to the increasing state of qubit.

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \quad (2.5)$$

where α_{00}^2, α_{01}^2, α_{10}^2 and α_{11}^2 are probability of qubits that have value *00, 01, 10* and *11*, respectively. Also, it must preserve the law of total probability, so their values must follow the equation 2.6.

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1 \quad (2.6)$$

Evaluation the value of multiple qubits is more complex than the single qubit. The order of qubits must be considered. If the value of first qubit is 0, the value of multiple qubits is evaluated by equation 2.7.

$$|\psi\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} \qquad (2.7)$$

From the equation 2.7, $\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}$ is a renormalize form in order to preserve the law of total probability. The double qubits have a significant state, which is Bell state or ERP pair. They have a correlation property, which is some relationship between two qubits. If the value in one of qubit is known, then the value of another qubit can be interpreted by using information of the known value qubit. Bell state is an important concept to define the Quantum teleportation phenomenon. Bell state is presented in equation 2.8.

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad (2.8)$$

B. *Quantum parallelism*

A quantum parallelism is a property of quantum computers which uses the advantage of the superposition property. When a unitary matrix operates with qubit, it will operate with all possibility of value. It can be explained with figure 2.

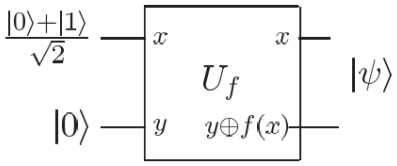

Figure 2: An example of a quantum circuit

From the figure 2, the quantum circuit has 2 inputs, which are $|x\rangle$ and $|y\rangle$. $f(x)$ is a controller qubit for input $|y\rangle$. The result of evaluation this circuit shows in equation 2.10.

$$|\psi\rangle = \frac{|0,f(0)\rangle + |1,f(1)\rangle}{\sqrt{2}} \quad (2.10)$$

In case of n-Qubit, the result shows in equation 2.11.

$$|\psi\rangle = \frac{1}{\sqrt{2^n}}\sum_x |x\rangle\,|f(x)\rangle \qquad (2.11)$$

An output depends on the function $f(x)$. Due to quantum parallelism property, a quantum computer can execute program faster than a classical computer. However, this property changes the way we design an algorithm. Many algorithms were designed for quantum computers such as Deutsch's algorithm or Deutsh-Jozsa algorithm [3].

III. QCL Simulator

QCL is one of the quantum computer simulator which is implemented in C++ language. The language of QCL has structured and it has hybrid mix of classical statements and quantum operations [4].

A. *Quantum Programming*

Most of the classical computers have high level language available, such as C, Java, Pascal, and so on. The high level languages are divided into 3 groups: logical, functional and procedural. A structured quantum programming [5] is an extended version of a procedural language. It extends the classical concept into the quantum concept.

B. *Hybrid architecture*

A hybrid architecture is a combination of classical computers and quantum computers. The inputs and outputs of

QCL simulator are classical bits. It behaves just like an ordinary input and output of a classical computer. However, the classical inputs are processed by the quantum program. QCL simulates a quantum machine to execute the quantum program. After that, the qubits are measured, so the qubits collapse into classical bits.

IV. Benchmark program

From the section 2 and 3, programing the quantum program is completely different from the classical program because of its behavior. Benchmark programs for QCL simulator should be the programs that are designed for quantum computer and coded in QCL.

In order to accelerate the QCL simulator, the simulator was inspected in details to analyze the data flow of program. The quantum version of the compact genetic algorithm [6] and Shor algorithm [7] were chosen as the benchmark programs to evaluate the QCL simulator.

Compact genetic algorithm

A compact genetic algorithm is a heuristic algorithm, which imitates the process of natural selection. It consists of three major steps.

1) The first step is to create the population. This step creates new data related to the old data's information. After that, the new data are adjusted with some rules.

2) The second step is evaluation. This step evaluates the data from first step. The data are evaluated with the fitness rule, which depends on the problem. The data that are closest to solution of the problem has the highest fitness.

3) The last step is determination. This step determines the data that have the highest possibility to be the solution of the problem.

Because of the probabilistic nature of executing a quantum algorithm, it is necessary to iterate the algorithm as many times as required to achieve a better accuracy. The higher number iteration returns more accurate solutions.

In order to take advantage of quantum computation, the compact genetic algorithm is modified. Due to the superposition property, the qubits are operated with all possible values. Thus, using the quantum gates and quantum bits is exponentially faster than a classical computer because the quantum parallelism property is applied.

V. Analysis

The QCL simulator was developed in C++ language. It operates in Linux operating system, so profiling was chosen as a tool to inspect the flow of program. The source code of QCL simulator was modified in order to perform profiling.

The QCL simulator executed the benchmark programs, which are compact genetic algorithm and Shor algorithm, with different qubit length and number of iteration. The profiler's result shows functions that spend more than 15% of the total execution time. They are these three functions:

1) *bitvec::bitvec(bitvec const&)*
2) *termlist::add(bitvec const&, std::complex<double>)*
3) *quSubString::unmap(bitvec const&) const*

In order to perform quantum operations, the QCL simulator specifies new data types such as bitvec, terminfo, qustate and so on. The most time consuming method is *bitvec::bitvec(bitvec const&)*, which is a constructor method. Therefore it should not be considered to be implemented in hardware circuits. The second function, *termlist::add(bitvec const&, std::complex<double> const&)*, were considered to be implemented on hardware circuits. Moreover, it calls other methods. The callee methods are implemented. A pseudo code of method *termlist::add(bitvec const&, std::complex <double> const&)* is described below.

```
void termlist::add(const bitvec& v, const complex& z) {
 Using value of input bitvec v perform hash
 //First part
 while(1)
  Store data termlist from hash in caller to pt
  if(pt doesn't have data) {
   if(caller size is smaller than its specification) {
    //Second part
    Increase caller size and call add method with same input
   } else {
    //Third part
    Create new termlist and store it to caller's hash at pt
   }
   return;
  }
  if(pt has bitvec same the input bitvec v) {
   //Fourth part
   Updating new data in pt with input z
   return;
  }
   //Fifth part
  Change hash function to get new hash index.
 }
}
```

The method is divided into 5 parts. The first part is the basic step which executes every time the method is called. It does a hash function, *hashfunct1*. Then enter a while loop. The second part is in the nested-if case. This part increases size of data storage and recreate it with new size. The third part is in the else case of nested-if. This part stores new data to the data storage. The fourth part is in if-case. This part updates old data in data storage which matches the input. The last part operates if and only if the other parts are not operated. This part does another hash, *hashfunct2*. Then it repeats the while loop.

VI. Design and Implementation

In order to implement the accelerator circuits, the high level data structures and operations must be changed. The methods coding in C++ were changed into hardware

description language. The accelerator circuits were designed to take an advantage of the hardware parallelism.

A. Flow chart

The flow chart of method was derived from the software code. The code is divided into 5 parts, so the flow chart has 5 states.

The flow chart of method is shown in figure 3.

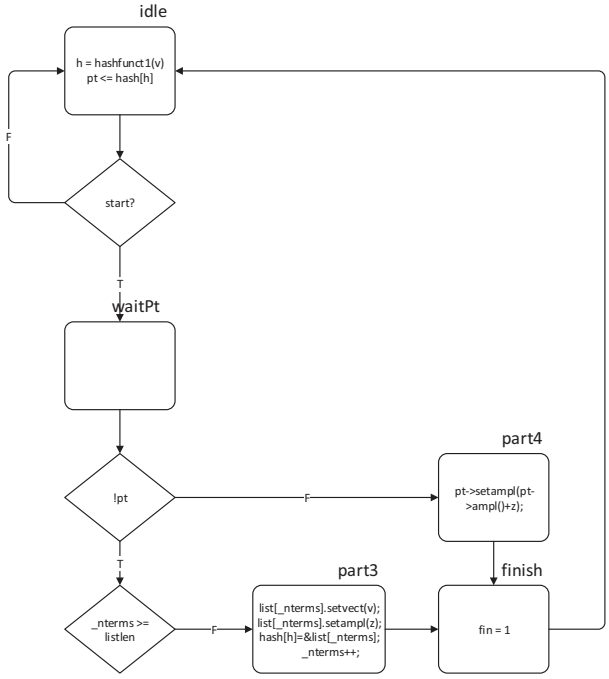

Figure 3: The modified flow chart of termlist::add(bitvec const&, std::complex<double> const&) method

The *termlist::add(bitvec const&, std::complex<double> const&)* module was designed to has 5 inputs and 2 outputs. The first input is clk signal, which controls the flip flop function. The second input is a start signal. The others are the inputs of the method: termlist, bitvec v and complex z. A block diagram of this module is in figure 4.

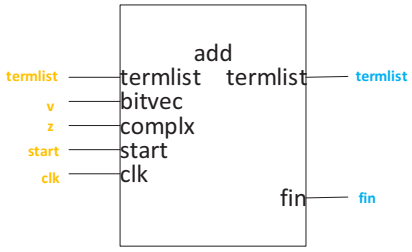

Figure 4: Block diagram of termlist::add(bitvec const&, std::complex<double> const&) method

1) State machine

The state machine was derived from the flow chart using Moore machine. It used 3 registers to store the current state.

2) Hashfunct module

The *hashfunct1* and *hashfunct2* have same root method, which is *bitvec::hashfunct()*. The difference of *hashfunct1* and *hashfunct2* is masking.

The *bitvec::hashfunct()* was implemented by combination circuits, so it instantly produces the result of its function.

B. Floating point unit

The *termlist::add(bitvec const&, std::complex<double> const&)* method uses complex data type, which is real number, so floating point unit must be designed because the tool for implementation of FPGA does not provide a floating point unit. The floating point unit was designed in the format IEEE 754 double precision standard.

In order to add floating point together, firstly, if the exponent bits of two numbers are not equal. The exponents must be aligned. The bigger exponent is decreased until it is equal the other. Then the fractional bits of the bigger number will change depend on how much exponent changed. After the exponent bits are equal, the add operation can be performed on fractional bits. The alignment of exponents requires an algorithm that finds the first bit that has value 1 in the exponent. The divide and conquer method is used to efficiently finding the first 1 value bit from the bit string. The bit string is divided into two substrings with equal length, the most significant substring and the least significant substring. Then, iteration the dividing with the most significant substring until it has only two bits. After that, two bits are compared with each other. If its value is both 0 check other two bits with in the same substring. If its value is both 1, the most significant bit is the first 1 value bit from bit string. If it is not equal, the higher one is the first 1 value bit from bit string. This method finds the first 1 value bit in O(log n).

The accelerator circuits were implemented in Verilog, which is one of the hardware description languages. The implementation is carried out on an FPGA board. Because of the limit number of input/output of the FPGA board, the data of two dimensional array, which are used in the simulator, cannot be fully implemented. To allow analysis of the speed up of the circuits, a partial implementation is done. Refer to the pseudo code of *termlist* method; the first part, the third part and the fourth part are implemented. These three parts consume 99% of the total execution time so they are good enough representative. The rest of the simulation is executed with the software.

VII. Experimental Result

To collect the experimental data, the interface between hardware circuits and the software of the QCL simulator is defined. The data going through this interface is collected. The hardware circuit is then tested offline (in separation from the simulator). The values through the interface are checked to validate the correct function of the accelerator circuits. To measure the speed up of the accelerator circuits, the number of cycles of the circuit (one clock) is compared with machine code instruction of the software (the assembly language of the PC machine) which is collected from the profile of the running

simulator. This assumption seems fair and allows the result to be independent of the speed of processor used in PC machine.

The accelerator circuits met the functional correctness. The comparison of the execution time between the accelerator circuits and the software is shown in Table 1.

Methods	Number of clock cycle	Number of machine instruction
termlist::add(const bitvec& v,const complex& z)	5	120
termlist::hashfunct1(const bitvec& v) const	instant	144
bitvec::hashfunct() const	instant	96
summary	**5**	**360**

Table 1: Comparing the accelerator circuits with the QCL simulator

VIII. Conclusion

From the section 7, it shows that accelerator circuits are much faster than the software. The accelerator is faster than the QCL simulator software 72 times.

The limit of this study is that it is driven by the profile of the simulator running benchmarks. Therefore the speed up is limited to the frequent operations in software. To fully explore the accelerator that aims to really speed up the simulator, perhaps the redesign of data structure should be considered. If the data structure is suitable for implementation is the hardware using RAM storing in the FPGA board, many other parts of simulation can be discovered that have high speed up potential. However, this approach will require rewriting the simulator.

References

[1] Tameem Albash, Walter Vinci, Anurag Mishra, Paul A. Warburton, Daniel A. Liar, Consistency tests of classical and quantum models for a quantum annealer, 2014.

[2] Michael A. Nielsen, Isaac L. Chuang, Quantum Computation and Quantum Information, 2011.

[3] David Deutsh and Richard Jozsa, Rapid soulutions of problems by quantum computation, Proceedings of the Royal Society of London A **439**: 553, 1992.

[4] Bernhard Umer. A Procedural Formalism for Quantum Computing. Master's Thesis, Department of Theoretical Physics, Technical University of Vienna, 1998.

[5] Bernhard Umer. Structured Quantum Programming. Ph.D. Thesis, Technical University of Vienna, 2003.

[6] Yingchareonthawornchai, S., Aporntewan, C., and Chongstitvatana, P., An Implementation of Compact Genetic Algorithm on a Quantum Computer, Int. Joint Conf. on Computer Science and Software Engineering, 2012, pp.131-135..

[7] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Computing 26, pp.1484-1509(1997).