Automatic Generation of Co-Processor for Simulation of Quantum Algorithms on FPGA

Calebe Conceição
Programa de Pós Graduação em Computação
Universidade Federal do Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
calebe.conceicao@inf.ufrgs.br

Ricardo Reis
Instituto de Informática – PPGC/PGMicro
Universidade Federal do Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
reis@inf.ufrgs.br

Abstract*—The simulation of quantum algorithms on classical computers is computationally hard, mainly due to the parallel nature of quantum systems. To speed up these simulations, some works have proposed to use programmable parallel hardware such as FPGAs, which considerably shorten the simulation execution time. It is needed a big effort to port the quantum algorithm to a FPGA. *We present a new tool that automatically generates an RTL description of a co-processor dedicated to simulate quantum algorithms using an FPGA.

Introduction

The search for new architectures and computational models is a way to find new improved solutions at the same time the fabrication technologies are approaching their physical limits [1] [2]. Many computational models have been proposed to replace or live together with silicon technologies [2][3] among which quantum computing is rather promising because it suggests considerable gains on computational power and appears to be compatible to all structure available today, which would soften its adoption as a new computational paradigm [4].

Quantum computing algorithms have been developed and tested through simulation using classical computers [5][6][7]. A problem is that such simulations are computationally hard due to the parallel nature of quantum world [8], since each operation of a quantum algorithm – which would be performed in parallel if executed in a quantum computer – must be scheduled to a sequential execution in a classical computer.

Alternatively, some works have used the intrinsic parallelism of programmable logic devices such as FPGAs to realize these simulations [9][10]. The gains in execution time of these approaches are about three orders of magnitude better than simulations on general purpose computers. The reason is that the needed hardware to perform small operations like sum and multiplications are simply replicated to execute in parallel, thus transporting the complexity of the problem from time domain to space domain.

This paper is organized as follows. In section 2 we present an brief overview of quantum computing. In section 3 we present how the co-processor is generated. Section 4 brings the results on logic cells usage using a quantum algorithms benchmark in comparison with related works. In section 4 we present the conclusions.

III - Details of the Tool

A. Graphical Interface

The graphical interface of the tool is based on the JQuantum simulator [7], an open source simulator of quantum circuits written in Java. Modifications were made to support the FPGA execution flow. A screen shot of the tool is presented on Fig. 1.

The circuit is designed using the circuit model and its schematic is displayed on central panel. The state vector of the system is displayed in the lower panel, where each amplitude is represented by the address of a quantum register. Colors represent complex values, according to the color map shown – a JQuantum original feature.

Once the designer describes its quantum algorithm using the circuit model, the tool automatically generates a synthesizable circuit description in the RTL level using SystemVerilog. Any third party tool can be used in order to synthesize and load it on FPGA.

B. Data Representation

The state vector is represented by a memory called quantum register. Each complex coefficient is stored in two registers, one for real and other for the imaginary part, whose size can be configured in 8, 16 or 32 bits. The real and imaginary part of a coefficient are stored on the same address. The data is represented in fixed point and complex arithmetic operators are

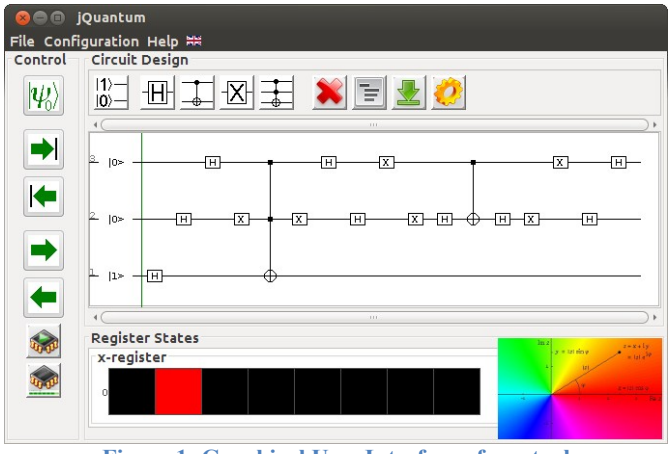

Figure 1: Graphical User Interface of our tool

parameterized. Considering the postulate of state space and postulate of evolution of quantum mechanics [8], only two bits are necessary to represent the signed integer part, and the remaining bits are used to represent the fractional part.

Since in quantum computing we can only prepare the initial state vector of a quantum algorithm in a pure state, just a string of 2^N bits (where N is the number of qubits) are sent as input in order to load the initial state vector. Each bit controls a set of 2^N multiplexers linked to each address to load the correspondent complex values on the quantum register.

Differently that in the initial state, the resulting state vector can be in a superposition state. So, it is necessary to output the entire quantum state register. The number of bits necessary to store the quantum register coefficients vector is given by $M2^{N+1}$, where M is the size of the mantissa.

C. Supported Quantum Gates

This tool supports Pauli X, Hadamard, CNot and Toffoli gates. They are implemented extending the idea of Network of Butterflies model, proposed on [11] just for Pauli X gate. By using this model it was possible to do a programmable gate implementation. The general idea of this model is shown on Fig. 5, using the Pauli X gate as example.

Since the transformation matrix of Pauli X gate has only complex constants 0 and 1, no sum or multiplication is necessary. The functionality of this single qubit gate, if applied to a system with only one qubit, consists on swapping the coefficients of the two states of the system. However, when it is applied in a compound system, although being applied to just one qubit, the functionality of the Pauli X gate shall be reflected on the entire system state, represented by its state vector.

As it can be seen on Fig. 2, it is necessary to select the right pairs of coefficients to swap their values on previous state vector. It is easily done by inverting the *target* bit of the binary representation of each coefficient index, where *target* is the number of the qubit to which the gate is applied. For instance, if the the Pauli X is applied to qubit 0 – ie. target is equals to 0 – the value to be stored on α_{00} of resultant state vector $|\psi_1\rangle$ comes from α_{01} of $|\psi_0\rangle$, and the value to be stored on α_{10} of $|\psi_1\rangle$ comes from α_{11} of $|\psi_0\rangle$, and so on.

In other words, to implement the Pauli X gate behavior, it is just necessary to identify the right pairs according to the target value, and swapping their values in the resultant state vector. This step can be done with multiplexers. The scheme is represented on Fig. 3.

On Fig. 3 is also shown the implementation scheme of the CNot gate. The behavior of this gate can be understood as an extension of Pauli X gate, but conditioned to the value of the qubit indicated by $ctrl_0$. In the implementation, if the $ctrl_0$ bit of binary representation of coefficient index is equals to one, the values of the selected pair of coefficients selected on first layer of multiplexers are exchanged; otherwise they are kept. The implementation of Toffoli gate is straightforward.

The Hadamard gate implementation scheme is quite similar to the Pauli X. The difference is that in this gate the data of $|\psi_0\rangle$ must be transformed before being written to $|\psi_1\rangle$ according to equations (1) and (2), derived from Hadamard matrix representation

$$\alpha_1 = \frac{\alpha_0 + \beta_0}{\sqrt{2}} \tag{1}$$

$$\beta_1 = \frac{\alpha_0 - \beta_0}{\sqrt{2}} \tag{2}$$

where α_0 and β_0 are a pair of coefficients of $|\psi_0\rangle$, and α_1 and β_1 are a pair of coefficients of $|\psi_1\rangle$. It is also the value of *target* qubit that controls the multiplexes to select the right pairs of coefficients for each coefficient of $|\psi_1\rangle$.

D. Quantum Processor

To simulate the quantum algorithm, our tool generates a single cycle co-processor that works at 50 MHz by default. Its block diagram is shown on Fig. 5. The ordered set of gates of a quantum algorithm is stored as instructions on Netlist Memory. The instructions have a format specified on Fig. 4. The operation of gates is controlled by the multiplexers according to the value specified in each field, as explained before.

The sequence of instructions must be loaded on the Netlist Memory. The Control Unit is responsible to address the Netlist

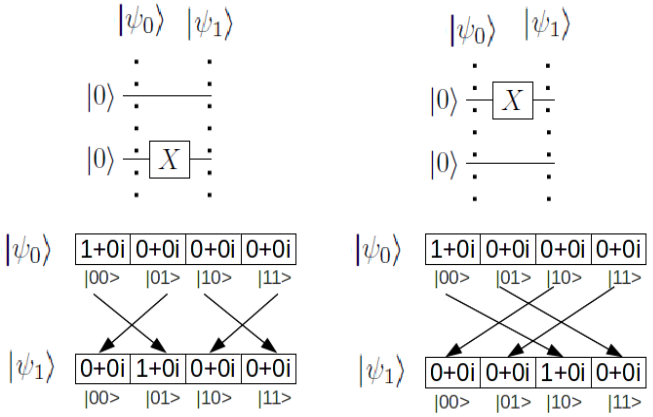

Figure 2: Pauli X gate behavior in a two qubits system

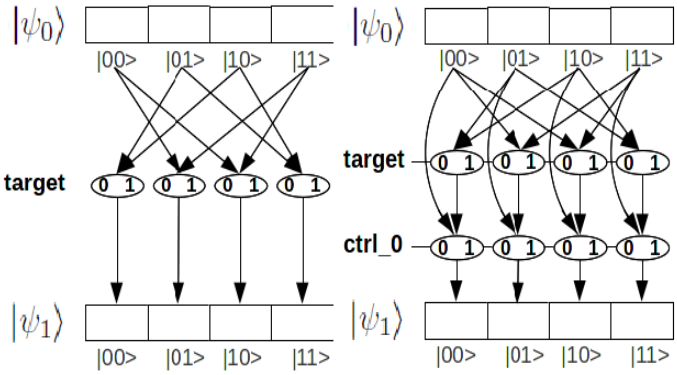

Figure 3: Pauli X (left) and CNot(right) implementation scheme

Memory neatly, starting when the *START* flag is sent, and to generate the *FINISHED* flag, that informs the end of computation.

The Quantum ALU module has just one instance of each gate, and it is responsible to select which output of them will be written on Quantum Register. With this approach only one quantum register is necessary to realize the computation.

The advantage of parallelism is taken when performing each complete operation in one clock cycle. If the same execution is done in an ordinary computer, it would take several processor cycles to perform each sum, multiplication and memory swap, for each pair of coefficients, sequentially.

IV – Results

The execution time of any algorithm performed on our co-processor is linear with the number of gates, since each gate performs its functionality in one clock cycle. However, as mentioned before, this approach of using FPGA just transfer the complexity from time domain to space domain. Therefore, the number of logic cells required to implement the co-processor increases exponentially with the number of qubits and the size of the mantissa, as can be verified on Table I. The Netlist Memory has a constant configurable size and therefore it is not shown.

Table I. LC usage of modules with the number of Qubits and Size of Mantissa

Unity	1 qubit	2 qubits	3 qubits			4 qubits	5 qubits
			8 bits	*16 bits*	*32 bits*		
Q. ALU	229	531	1402	3526	9408	2982	6869
Q. Reg	32	64	128	256	512	256	512

Mantissa fixed on 8 bits where it is not mentioned. Q. Reg values are expressed in number of flip-flops

On the other hand, our solution is invariant to the number of gates in the algorithm description, differently of the solution proposed on [10], that depends on that. The results on Table I shows that Quantum ULA is the main responsible to the exponential increase of the size of the system. Also, it is better than [9] in terms of memory usage, which includes a new quantum register after each quantum gate.

We also measured the logic cells usage for circuits of a quantum algorithm benchmark of reversible circuits avaliable in [12] and used as metrics in previous work. In order to fairly compare our results, we kept in the Quantum ALU only instances of quantum gates that are used in the circuit description. Additionally, we also reduce the size of the Netlist Memory to hold up only the number of gates in the quantum algorithm description. The logic cell usage of the generated co-processor for 8 and 16 bits in the mantissa is shown on Table II. The benchmark circuits are ordered by the number of qubits.

The results in logic cells of our automatically generated RTL description is comparable to the best results available in literature [10], as can be seen on Table II. There is almost no variation when increasing the mantissa of a circuit, which differs from results shown on Table I. It is because no circuit of the benchmark uses Hadamard gate, whose size is the most sensible this variation.

Data	$ctrl_1$	$target$	$ctrl_0$	$opcode$
Size (bits)	$\lceil log_2 N \rceil$	$\lceil log_2 N \rceil$	$\lceil log_2 N \rceil$	*2*

Figure 4 Instruction format (N is the number of qubits)

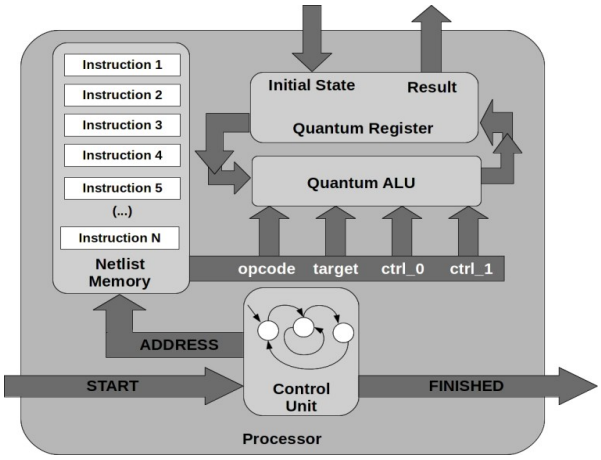

Figure 5: Quantum processor generic block diagram

Table II. Logic Cells Usage for Benchmark Circuits

Circuit	Ours		[10]		[9]	
	8 bits	*16 bits*	*8 bits*	*16 bits*	*8 bits*	*16 bits*
3_17tc	150	150	24	24	960	1728
ham3tc	140	140	24	24	800	1440
rd32	191	191	48	48	1280	2304
hwb4-11-23	230	230	64	64	3520	6336
xor5d1	416	409	128	128	2560	4608
Mod5d1	571	573	224	224	5120	9216
greycode6	903	905	320	320	6400	11520
rd53d2	4019	4019	3072	3072	-	-

The results of [9] are shown on paper [10].

To evaluate the Hadamard gate scalability, we plot on Fig. 6 the logic cell usage of this gate for systems with up to 4 qubits, with 8, 16 and 32 bits on mantissa. Looking at this, it is straightforward to conclude that this gate is the main responsible for the Quantum ALU exponential increase. This is due to the use of multipliers and adders in the implementation of this gate, as mentioned on previous section.

Lastly, we tried to discover the largest co-processor we are able to fit in our FPGA, an Altera Cyclone II EP2C35F672C6 chip, that contains about 35 thousand logic elements [13]. Fig. 7 illustrates the exponential shape of the curve representing the logic cell usage in function of the number of qubits. However, by using our tool, one can generate a co-processor for a fixed number of qubits (up to 6 qubits on this FPGA platform) and mantissa, and synthesize it just once for any circuit he want to simulate. Its is an advance compared to the others solutions, since the compilation time can also increase exponentially.

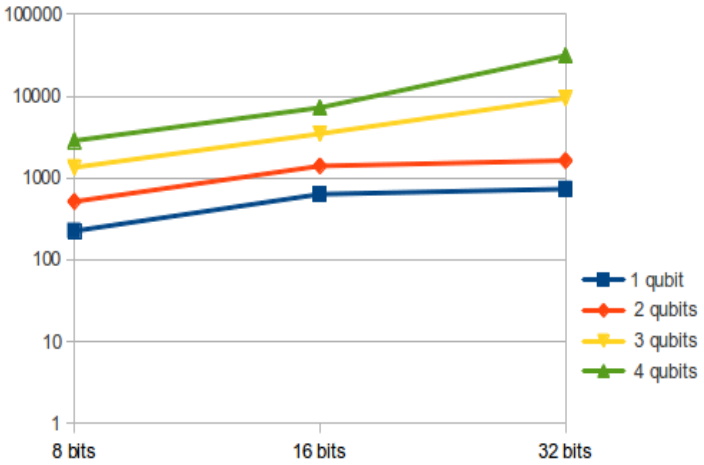

Figure 6: Logic Cell usage of Hadamard gate

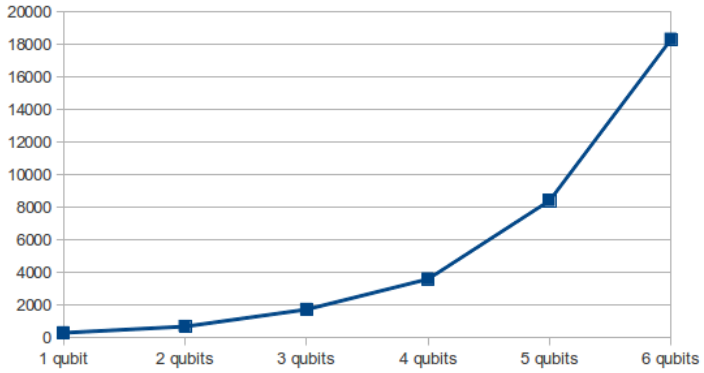

Figure 7: Logic Cells with the number of qubits (mantissa = 8)

V – Conclusions

We presented a tool that automatically generates a library independent RTL description of a co-processor for simulation of quantum algorithms on FPGA. The results in logic cell usage are comparable to the best manually generated descriptions found on literature. Besides that, our method is programmable, which makes the logic cell scaling independent of the number of quantum gates in the quantum algorithm and eliminates the need of re-synthesis the RTL description if the number of qubits is kept.

This approach of using FPGA on simulation of quantum algorithms just transfers the complexity of this problem from time domain to space domain, and therefore the space scales exponentially. However, if one could deal with scaling on space domain, quantum algorithms would be executed efficiently in time even using classical hardware.

References

[1] J. Powell, “The Quantum Limit to Moore’s Law,” Proceedings of the IEEE, vol. 96, no. 8, pp. 1247 –1248, aug. 2008. DOI 10.1109/JPROC.2008.925411

[2] S. Bampi and R. Reis, "Challenges and Emerging Technologies for System Integration beyond the End of the Roadmap of Nano-CMOS", VLSI-SoC: Technologies for Systems Integration, vol. 360, pp.21-33, Springer Berlin Heidelberg, 2011. DOI 10.1007/978-3-642-23120-9_2

[3] P. Warren, “The future of computing - new architectures and new technologies,” Nanobiotechnology, IEE Proceedings -, vol. 151, no. 1 pp. 1 – 9, 5 2004. DOI 10.1049/ip-nbt:20030876

[4] E. Rieffel and W. Polak, “An Introduction to Quantum Computing for Non-Physicists,” ACM Comput. Surv., vol. 32, no. 3, pp. 300–335, 2000. DOI 10.1145/367701.367709

[5] I. Karafyllidis, “Quantum computer simulator based on the circuit model of quantum computation,” Circuits and Systems I: Regular Papers, IEEE Transactions on, vol. 52, no. 8, pp. 1590 – 1596, aug. 2005. DOI 10.1109/TCSI.2005.851999

[6] B. Butscher and H. Weimer, “Libquantum: the c library for quantum computing and quantum simulation,” 2004–2011. [Online]. Available: http://www.libquantum.de/

[7] A. de Vries, “JQuantum - a Quantum Computer Simulator,” 2004–2011. [Online]. Available: http://jquantum.sourceforge.net/

[8] M. A. Nielsen and I. L. Chuang, “Quantum Computation and Quantum Information”, Bookman, 2000. ISBN-10: 0521635039

[9] A. Khalid, Z. Zilic, and K. Radecka, “Fpga emulation of quantum circuits,” in Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on, 11- 13 2004, pp. 310 – 315. DOI 10.1109/ICCD.2004.1347938

[10] M. Aminian, M. Saeedi, M. Zamani, and M. Sedighi, “Fpga-based circuit model emulation of quantum algorithms,” in Symposium on VLSI, 2008. ISVLSI ’08. IEEE Computer Society Annual, 7-9 2008, pp. 399 –404. DOI 10.1109/ISVLSI.2008.43

[11] G. Negovetic, M. Perkowski, M. Lukac, A. Buller, “Evolving Quantum Circuits and an FPGA-Based Quantum Computing Emulator”, International Workshop on Boolean Problems, 2002.

[12] D. Maslov, “Reversible Logic Synthesis Benchmarks”, Available: http://webhome.cs.uvic.ca/~dmaslov/

[13] Altera, “DE2 Development and Education Board - User Manual”, PDF file, Altera Corporation, 2001.

[14] P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring”. In:Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. [S.l.: s.n.],1994. p. 124 –134. DOI 10.1109/SFCS.1994.365700

[15] L. K. Grover, “A fast quantum mechanical algorithm for database search”. In: STOC ’96: Proc. the 28th annual ACM symposium on Theory of computing. New York, NY, USA: ACM, 1996. p. 212–219. DOI 10.1145/237814.237866