COMEC 2nd Edition Volume I

Simulando o paralelismo quântico em CPU e GPU utilizando a biblioteca LibQuantum

Samuel da Silva Feitosa[1] **, João Augusto da Silva Bueno**[2]

[1]Programa de Pós Graduação em Informática
Universidade Federal de Santa Maria (UFSM)
Caixa Postal 97105-900 – Santa Maria – RS – Brazil

[2]Departamento de Informática
Instituto Federal de Santa Catarina (IFSC)
Caixa Postal 89500-000 – Caçador – SC – Brazil

Abstract. *Despite all computational advances of the last decade, there are still hard to solve classical problems in a efficient way using regular computers. Researchers believe that through a quantum computer, some of that problems could be solved in few hours or even few minutes. However, considering the unavailability of quantum computers, several simulators have been proposed. In this paper, we discuss the* LibQuantum *library as a way to simulate the quantum parallelism through multi-core processors and GPGPUs.*

Resumo. *Apesar de todos os avanços computacionais da última década, ainda existem problemas clássicos que são difíceis de resolver de forma eficiente em computadores normais. Pesquisadores acreditam que, através de um computador quântico, alguns desses problemas poderiam ser resolvidos em algumas horas ou mesmo em alguns minutos. Entretanto, devido a indisponibilidade de computadores quânticos, diversos simuladores têm sido propostos. Neste artigo, é discutida a biblioteca* LibQuantum *como uma maneira de simular o paralelismo quântico através do uso de processadores multi-core e GPGPUs.*

1. Introdução

Computação quântica [Nielsen and Chuang 2011] é uma área de pesquisa recente que considera dispositivos desenvolvidos em escalas muito pequenas, as quais são governadas pelas leis da mecânica quântica. Este campo de pesquisa tenta encontrar formas de manipular os efeitos quânticos para melhorar o processamento da informação utilizando-se de novas abordagens. Pesquisadores da ciência da computação quântica afirmam que computadores quânticos serão capazes de executar certas tarefas computacionais em menos passos que qualquer computador convencional. Esta afirmação é justificada pois os algoritmos quânticos disponíveis tem a capacidade de manipular fenômenos físicos que não estão disponíveis em computadores convencionais [Williams 2008].

Uma área de pesquisa interessante em computação quântica é a simulação dos fenômenos da mecânica quântica em computadores convencionais [Butscher and Weimer 2013, Belkner 2012, Gheorghiu 2016, Microsoft 2016, Google 2016], que além de facilitar a compreensão dos conceitos de computação quântica em geral, permite também a criação e simulação de novos algoritmos quânticos. É importante salientar que devido à indisponibilidade de *hardware* quântico, o estudo

e desenvolvimento de aplicações para computação quântica tem sido realizado estritamente pela especificação matemática das computações ou por meio de ferramentas de simulação [Yanofsky and Mannucci 2008].

Levando em consideração que a computação quântica apresenta o fenômeno do paralelismo quântico [Gruska 2000], se faz necessário que os simuladores modelem estas propriedades. Para este trabalho, optou-se por utilizar a *LibQuantum* [Butscher and Weimer 2013], uma biblioteca desenvolvida na linguagem C, com foco especial na simulação da mecânica e da computação quântica. Esta biblioteca já apresenta uma implementação de paralelismo através do framework *OpenMP*, possibilitando explorar as capacidades de processadores *multi-core*. Todavia, neste trabalho foi adicionado o suporte ao processamento em GPGPUs, para, ao final, realizar uma comparação entre as duas abordagens.

Este artigo está dividido da seguinte forma: conceitos básicos de computação quântica, que são apresentados na seção 2. Uma breve explanação a respeito da biblioteca *LibQuantum* é apresentada na seção 3. Na seção 4 são discutidos os aspectos relacionados à implementação e os resultados obtidos nas execuções realizadas. Na seção 5 são apresentados alguns trabalhos relacionados. Por fim, na seção 6 são apresentadas as conclusões e trabalhos futuros.

2. Computação Quântica

A unidade básica de informação na computação clássica é o *bit*, que representa o sistema físico clássico binário, sendo capaz de representar apenas dois estados (*true* ou *false*, 0 ou 1). Qualquer informação é descrita como uma combinação de sequências de *bits*.

Na computação quântica, a unidade básica de informação é chamada de *quantum bit* ou *qubit*. O *qubit* apresenta uma diferença essencial se comparado com o *bit*, pois ele não está restrito aos dois estados básicos do *bit* clássico, podendo estar efetivamente em ambos os estados (0 e 1) ao mesmo tempo [Nielsen and Chuang 2011]. Vários centros de pesquisa têm estudado maneiras de manipular partículas que sejam capazes de fornecer as características quânticas, porém, ainda existem diversos desafios na manipulação física de elementos em escala microscópica.

O campo de estudo teórico da computação quântica, define matematicamente um *qubit* como sendo um vetor [1]:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

onde cada uma de suas posições armazena a *amplitude de probabilidade* α e β de cada um dos estados básicos. As amplitudes de probabilidade são *números complexos*, tal que $|\alpha|^2 + |\beta|^2 = 1$.

Intuitivamente, pode-se pensar no *qubit* como sendo 0, 1 ou ambos os estados simultaneamente, tendo um coeficiente numérico que determina a probabilidade de cada estado puro.

Qualquer outro estado com valores diferentes para α e β representa uma *superposição quântica* de $|0\rangle$ e $|1\rangle$. Estes estados em *superposição* fornecem à computação quântica uma característica chamada *paralelismo quântico*. Essencialmente,

[1] Na notação de Dirac.

devido à *superposição* de estados, um *qubit* pode assumir os valores 0 e 1 ao mesmo tempo. Esta propriedade é explorada nos algoritmos quânticos, que podem até obter um crescimento exponencial na sua velocidade, devido à característica de manipular várias possibilidades em paralelo.

Os *bits* clássicos podem ser examinados para determinar seu valor atual (0 ou 1), e isto é feito a todo momento nos computadores, manipulando o conteúdo da memória. No caso dos *qubits* é impossível visualizar seus valores para determinar seu estado atual (amplitudes α e β) sem interferir com o sistema. Uma leitura do estado quântico realiza uma *operação de medida*.

Na computação quântica pode-se efetuar dois tipos de operações: *operações de medida* e *transformações unitárias*. A operação de medida está relacionada à maneira de obter informações de um estado quântico. Transformações unitárias referem-se a operações que transformam o atual estado quântico em outro, como ocorre na aplicação de uma função na computação clássica.

A computação quântica também difere da computação clássica por ser probabilística, ou seja, a operação de medida trabalha sobre as amplitudes de probabilidade de um estado quântico. Quando uma medição é executada, acontece um colapso nas amplitudes de probabilidades e apenas um dos estados básicos é retornado, como $|0\rangle$ ou $|1\rangle$. Em outras palavras, após uma medição, o *qubit* fica em um estado conhecido, e as amplitudes de probabilidade são destruídas. A operação de medida geralmente é utilizada para obter as informações após o processamento de um algoritmo quântico.

De forma similar ao modo de processar informações em sistemas clássicos, na computação quântica um algoritmo é projetado como uma série aplicações de transformações unitárias, ou *portas quânticas*. Estas portas, aplicadas aos *qubits*, modificam seus valores de modo a transformar um estado quântico inicial na saída desejada.

3. LibQuantum

LibQuantum [Butscher and Weimer 2013] é uma biblioteca desenvolvida na linguagem C e criada para a simulação da mecânica quântica, com um foco especial para a computação quântica. Iniciou-se como um simulador de computador quântico puro, o suporte para simulação quântica em geral foi recentemente adicionado, tornando a biblioteca ainda mais funcional. Esta biblioteca provê uma estrutura para registradores quânticos (a memória de um computador quântico) e operações para a manipulação deste registrador.

Dentre as principais características, pode-se citar:

- É possível a simulação de algoritmos quânticos arbitrários.
- Alta performance (explorando o paralelismo) e baixo consumo de memória.
- Suporte a de-coerência, permitindo uma simulação mais realística.
- Interface para correção de erros quânticos.

O paralelismo incluído nesta biblioteca considera o uso API *OpenMP*, possibilitando a execução em paralelo de diversos trechos de código presentes na implementação da *LibQuantum*. Considerando a estrutura paralela já definida, neste trabalho adicionou-se a possibilidade de execução dos blocos paralelos também em *GPU*.

4. Implementação e Resultados

A abordagem adotada neste trabalho compreende o estudo do funcionamento básico da biblioteca *LibQuantum*, realizando testes e visualizando os algoritmos quânticos passíveis de execução na mesma. Nesse contexto, percebe-se que a biblioteca foi desenvolvida de modo bem modularizado, o que traz uma organização de código considerável, porém, para efeitos de paralelização acaba dificultando o processo. Aprofundando os conhecimentos relativos à implementação já existente, notou-se a exploração do paralelismo através da API *OpenMP*, possibilitando a execução em paralelo através de *CPUs multi-core*.

De posse destas informações, realizou-se uma série de execuções para verificar a possibilidade e a escalabilidade de paralelismo através dos mecanismos já existentes. Neste *benchmark*, considerou-se a utilização dos algoritmos clássicos da computação quântica oferecidos como exemplo de uso pela própria biblioteca *LibQuantum*, sendo o algoritmo de *Shor*, que em teoria deve realizar a fatoração de números com complexidade O(log_n) e o algoritmo de *Grover*, que tem o objetivo de realizar buscas em bases de dados não estruturadas, também apresentando um ganho considerável em relação ao seu respectivo algoritmo clássico.

Os tempos coletados para o algoritmo de *Shor* considerando execução serial, com duas threads e com 4 threads (simuladas) são apresentados na Figura 1.

	Tempo em segundos		
Número Utilizado	Serial	2 Threads	4 Threads
100	0,55	0,35	0,37
200	3,07	1,83	1,92
300	9,06	5,23	5,29
400	16,98	9,96	10,31
500	18,16	10,77	12,09
600	57,69	41,81	45,76
700	58,94	41,74	44,78
800	113,33	93,34	98,92
900	123,87	100,40	106,25
1000	117,85	97,17	115,24

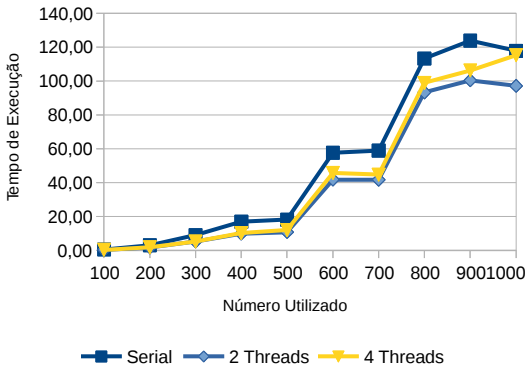

Figura 1. Execução do algoritmo de *Shor* utilizando *OpenMP*.

Os tempos coletados para o algoritmo de *Grover* considerando execução serial, com duas threads e com 4 threads (simuladas) são apresentados na Figura 2.

Considerando estas execuções, percebeu-se que a biblioteca *LibQuantum* possui, de fato, a possibilidade de explorar a paralelismo, como na comparação dos testes de execução serial e paralela via *OpenMP*.

4.1. Migração da *LibQuantum* para execução em *GPU*

Durante as implementações foram realizadas várias simulações e transformações na codificação da biblioteca *LibQuantum*, para que a mesma executasse paralelamente em *GPU*, almejando assim resultados relevantes.

Número Utilizado	Tempo em segundos		
	Serial	2 Threads	4 Threads
10k	2,07	1,91	2,40
20k	6,18	5,64	7,35
30k	5,91	5,47	7,39
40k	18,27	16,79	21,76
50k	18,00	16,67	21,40
60k	18,17	16,41	21,29
70k	63,82	59,18	79,27
80k	65,56	60,76	82,02
90k	64,20	58,21	78,11
100k	65,69	59,87	79,56

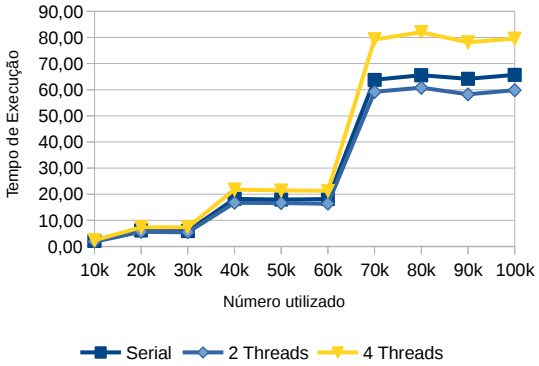

Figura 2. Execução do algoritmo de *Grover* utilizando *OpenMP*.

4.1.1. Utilização da API *OpenAcc*

A primeira abordagem para utilização de *GPGPU* com a *LibQuantum* foi através da utilização das diretivas disponibilizadas através da API *OpenAcc* [OpenAcc 2016], que é um padrão para simplificar o desenvolvimento de programação paralela com *CPU/GPU*.

Do mesmo modo utilizado pela *OpenMP*, os programadores apenas realizam anotações no código fonte, indicando áreas que podem ser *aceleradas*, e o compilador fica responsável por fazer a geração de código para utilizar os recursos dos equipamentos físicos.

As diretivas *OpenAcc* são relativamente bem documentadas, e a partir de uma leitura abrangente da documentação, compreendeu-se o funcionamento das mesmas. Realizou-se a codificação, criando uma referência aos ponteiros internamente na função chamadora e, posteriormente copiando esta referência interna para a memória da *GPU*. Feito isso, realizou-se a compilação e execução dos seguintes códigos:

```
1  #ifndef _OPENMP // Tratamento para OpenACC
2    MAX_UNSIGNED *reg_state = reg->state;
3    int reg_size = reg->size;
4  #endif
5    // Codigos para tratamento com OpenMP ...
6  #else
7  #pragma acc parallel present_or_copy(reg_state[:reg_size]) {
8    #pragma acc loop
9    for(i=0; i<reg_size; i++) {
10     if(reg_state[i] & ((MAX_UNSIGNED) 1 << control1)) {
11       if(reg_state[i] & ((MAX_UNSIGNED) 1 << control2)) {
12         reg_state[i] ^= ((MAX_UNSIGNED) 1 << target);
13       }
14     }
15   }
16 }
```

Verificou-se que os códigos estavam sendo executados corretamente e realizando o processamento em *GPU*. A comprovação foi através do uso da ferramenta disponibilizada pela *NVidia*, chamada *nvidia-smi*.

Com o código implementado e funcional, foi iniciada uma nova bateria de testes, de modo a comparar os efeitos da abordagem através do processamento em *GPU*. Percebeu-se, logo no primeiro teste, que os códigos que utilizavam *GPU*, levavam tempo

muito superior aos mesmos trechos de códigos executando em *CPU*, incentivando a realizar novas pesquisas, buscando saber o que estava ocorrendo.

Realizaram-se diversos testes, objetivando a coleta do tempo de execução de cada chamada, contabilização de execuções, etc. Como a *OpenAcc* é uma biblioteca que reconhece diretivas de compilação, muitas das codificações que ela implementa, são imperceptíveis ao programador, atrapalhando a instrumentação do código, uma vez que não é possível separar os passos de execução.

Após uma instrumentação manual do código, ficou claro que:

1. As funções com códigos sendo processados em *CPU* eram chamadas muitas vezes.
2. Como efeito de implementação das cópias dos ponteiros internos das *structs* para uma variável local, os dados eram copiados para *GPU* em cada chamada.
3. Os *loops* de execução paralela eram relativamente pequenos, o que, em geral não compensava o tempo de cópia dos dados.

Deste modo, partiu-se para a implementação através da *API CUDA*, que permite ao programador maior controle sobre as instruções a serem executadas em *GPU*.

4.1.2. Utilização da *API CUDA*

CUDA (*Compute Unified Device Architecture*) [NVidia 2016] é uma biblioteca que contém diversas funções para computação de propósito geral em *GPU*. É baseada na linguagem *C* padrão e possui uma *API* simples para manipular dispositivos, memória, etc. Com esta *API*, manipula-se diretamente a alocação, cópias e gerenciamento de dados na memória, criação de núcleos paralelos de execução, comunicação paralela, sincronização e operações atômicas.

Esta biblioteca trabalha com a terminologia de códigos *host* e *device*, principalmente para tratar da execução e do gerenciamento de memória do computador/servidor e do dispositivo *GPU*. Ao comparar a utilização da *API CUDA* com a *OpenAcc* na execução de diversos exemplos, pôde-se notar que através do uso de *CUDA* o desenvolvedor tem um maior controle sobre a alocação de memória, cópia dos dados do *host* para o *device*, execução do código, etc.

Na implementação da biblioteca *LibQuantum* utilizando a API *CUDA*, seguiu-se os mesmos princípios da implementação já realizada através de *OpenAcc*, migrando-se as mesmas funções paralelas para rodar no dispositivo *GPU*.

A primeira etapa foi compilar os códigos atuais com o compilador fornecido pelo fabricante (*nvidia*). Notou-se que o compilador gera código *C++*, e para evitar erros do *linker*, foi necessário fazer uma alteração nos códigos adicionando a diretiva *extern C* para todos os arquivos *headers* incluídos no desenvolvimento da biblioteca *LibQuantum*. Deste modo, a compilação ocorreu sem apresentar erros.

Na sequência, partiu-se para a fase de codificação utilizando *CUDA*. Primeiramente foi criada a função *kernel* para ser executada em *GPU*, conforme o código que segue:

```
1  extern "C"
2  __global__ void cuda_toffoli(MAX_UNSIGNED *state, int *control1,
3                               int *control2, int *target) {
4    int index = threadIdx.x + blockIdx.x * blockDim.x;
5
6    if (state[index] & ((MAX_UNSIGNED) 1 << *control1)) {
7      if (state[index] & ((MAX_UNSIGNED) 1 << *control2)) {
8        state[index] ^= ((MAX_UNSIGNED) 1 << *target);
9      }
10   }
11 }
```

E após isto, modificou-se a função disponibilizada pela biblioteca *LibQuantum* de modo a transferir os dados a serem processados em *GPU*, e também solicitar a execução e coletar o resultado do processamento paralelo. O código abaixo foi responsável por este processo.

```
1  extern "C"
2  void quantum_toffoli(int control1, int control2,
3                       int target, quantum_reg *reg) {
4    MAX_UNSIGNED *dev_state;
5    int *dev_control1, *dev_control2, *dev_target;
6    int state_size = reg->size * sizeof(MAX_UNSIGNED);
7    // Alocando memoria para GPU
8    cudaMalloc( (void **) &dev_state, state_size);
9    cudaMalloc( (void **) &dev_control1, sizeof(int));
10   ...
11   // Copiando os dados para a GPU
12   cudaMemcpy(dev_state, reg->state, state_size, cudaMemcpyHostToDevice);
13   cudaMemcpy(dev_control1, &control1, sizeof(int), cudaMemcpyHostToDevice);
14   ...
15   // Executando a funcao em GPU
16   cuda_toffoli<<< (reg->size + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK,
17          THREADS_PER_BLOCK >>> (dev_state, dev_control1, dev_control2, dev_target);
18   // Copiando a resposta para o Host
19   cudaMemcpy(reg->state, dev_state, state_size, cudaMemcpyDeviceToHost);
20   // Liberando memória alocada em GPU
21   cudaFree(dev_state);
22   ...
```

Além da codificação, realizou-se uma instrumentação total do código, de modo a coletar o tempo exato da transferência de dados do *HOST* para *GPU* e da *GPU* para *HOST*. Os resultados coletados a partir códigos após a implementação das mesmas em paralelo com a execução em *GPU* (considerando o tempo de execução completo, a transferência de dados e o tempo de execução do código paralelo) para o algoritmo de *Shor* e de *Grover* são apresentados na Figura 3.

Em ambos os algoritmos podemos visualizar que o tempo de execução foi muito maior que a soma da cópia dos dados e da execução paralela, justificando-se pelo uso de inúmeras outras funções que não foram paralelizadas e também não foram contabilizadas neste trabalho. Além disso, percebe-se que os dados de transferência das informações do *host* para *GPU* consomem um tempo muito superior ao tempo de execução no dispositivo, sendo que nos testes realizados, nenhum foi capaz de apresentar desempenho superior à da execução serial em *CPU*.

Algoritmo de Shor

Número Utilizado	Tempo em segundos: Execução Completa	Transf. de Dados	Execução Paralela
100	9,29	8,11	0,10
200	24,58	18,23	0,14
300	58,42	41,00	0,22
400	87,41	52,76	0,20
500	91,28	55,47	0,21
600	213,73	121,71	0,32
700	217,11	125,02	0,31
800	382,44	195,89	0,35
900	402,52	210,23	0,38
1000	396,68	205,56	0,36

Algoritmo de Grover

Número Utilizado	Tempo em segundos: Execução Completa	Transf. de Dados	Execução Paralela
10000	9,77	0,03	0,00040
20000	25,28	0,20	0,00045
30000	24,55	0,04	0,00047
40000	66,21	0,06	0,00048
50000	66,25	0,22	0,00050
60000	65,98	0,22	0,00052
70000	165,01	0,09	0,00051
80000	166,23	0,09	0,00049
90000	164,12	0,11	0,00053
100000	164,42	0,09	0,00047

Figura 3. Tempo de execução em *GPU* dos algoritmos de *Shor* e de *Grover*.

4.1.3. Comparação *GPU / CPU*

Finalizando, comparou-se somente entre o tempo real de processamento do *loop* paralelo na execução em *GPU* e em *CPU*. Os resultados do algoritmo de *Shor* são apresentados na Figura 4.

Número Utilizado	Tempo em segundos: Execução em GPU	Execução em CPU
100	0,10	1,00
200	0,14	5,28
300	0,22	17,71
400	0,20	31,48
500	0,21	37,42
600	0,32	92,39
700	0,31	93,10
800	0,35	173,82
900	0,38	183,69
1000	0,36	187,35

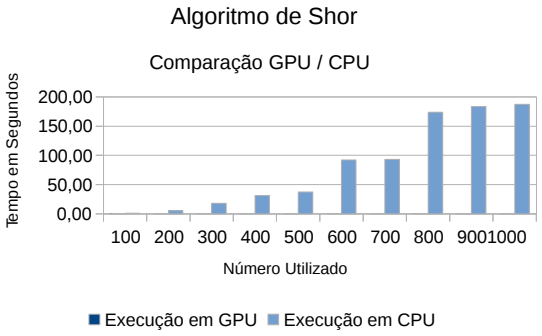

Figura 4. Comparação das execuções em *GPU* e *CPU* para o algoritmo de *Shor*.

No gráfico apresentado na Figura 4 nota-se a grande diferença entre o tempo de execução somente do código executado em *GPU* quando comparado com a execução em *CPU*.

Já na Figura 5 são apresentados os dados referentes à execução do algoritmo de *Grover*, onde também notou-se comportamento similar ao ocorrido com o algoritmo de *Shor*. Em ambos os casos, a execução do código paralelo em *GPU* fica quase imperceptível no gráfico comparado à execução em *CPU*.

A partir dos dados apresentados nesta seção, foi possível perceber que apenas a execução dos códigos em *GPU* apresenta um desempenho imensamente superior ao trecho de código executado em *CPU*. Isto se explica pelo fato da *GPU* possuir um número muito maior de núcleos de processamento.

	Tempo em segundos	
Número Utilizado	Execução em GPU	Execução em CPU
10k	0,00040	0,00723
20k	0,00045	0,01556
30k	0,00047	0,01534
40k	0,00048	0,03300
50k	0,00050	0,03253
60k	0,00052	0,03286
70k	0,00051	0,06878
80k	0,00049	0,06963
90k	0,00053	0,06874
100k	0,00047	0,06964

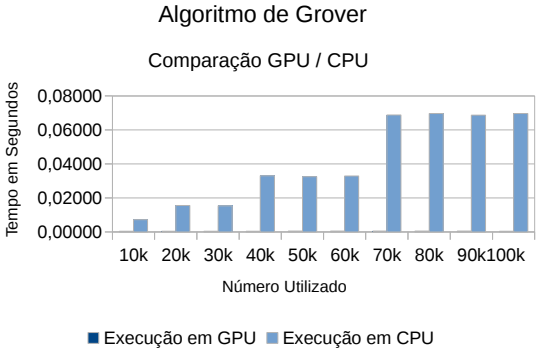

Figura 5. Comparação das execuções em *GPU* e *CPU* para o algoritmo de *Grover*.

5. Trabalhos Relacionados

Os simuladores de computação quântica permitem executar sistemas quânticos que são complexos de estudar em laboratório e até mesmo difíceis de modelar em um supercomputador. Neste sentido, os simuladores tem um propósito especial de prover mecanismos para trabalhar com problemas físicos específicos.

Considerando a importância para estes tipos de estudos, diversos simuladores de computação quântica foram propostos, onde, além da biblioteca *LibQuantum*, podem-se citar as bibliotecas *Eqcs* [Belkner 2012], *Quantum++* [Gheorghiu 2016], os simuladores oferecidos pela Microsoft [Microsoft 2016] e Google [Google 2016], dentre outros.

Além das bibliotecas de simulação, há também um esforço dos pesquisadores que apresentam linguagens de programação para computação quântica, que, em geral, também podem ser utilizadas para simulação de algoritmos [Ömer 1998, Sanders and Zuliani 1999, Selinger 2008, van Tonder 2004, Vizzotto et al. 2013, Feitosa et al. 2015].

6. Conclusões

Neste trabalho foi realizada uma implementação em forma de extensão da biblioteca *LibQuantum*, onde foram adicionadas as possibilidades de execução dos algoritmos quânticos também em *GPU*, além das características já implementadas pela biblioteca com relação ao paralelismo em computadores *multi-core*. Além disso, foram realizadas diversas execuções dos algoritmos quânticos (*Shor* e *Grover*) a fim de comparar os desempenhos através das diferentes abordagens.

Inicialmente foram realizados diversos testes para verificar a capacidade de paralelismo e escalabilidade na melhora de desempenho através dos mecanismos já implementados na biblioteca. Constatou-se que, de fato, a implementação em *CPUs multi-core* apresenta melhoria de desempenho em comparação com a execução serial. Finalmente, desenvolveu-se a extensão que implementa o mecanismo de paralelismo em *GPU*, e realizou-se a comparação com a implementação já existente.

No caso das execuções em *GPU*, conforme demonstrado nos resultados, foi percebido que realmente a execução destes blocos de código apresentam desempenho muito

superior à execução *multi-core*, quando não são consideradas as diversas trocas de contexto entre a memória principal e a memória do dispositivo *GPU*. Porém, devido ao grande número de chamada das funções responsáveis pelo processamento paralelo, diversas trocas de contexto se fazem necessárias, o que acaba por prejudicar o desempenho da execução neste tipo de dispositivo.

Como trabalho futuro, seria interessante implementar um contexto global para trabalhar com os dispositivos *GPU* de modo que não se faça necessário as diversas trocas de contexto, uma vez que o estado quântico é um estado global e é modificado durante toda a execução do algoritmo. Acredita-se que, através desta abordagem é possível melhorar suficientemente o desempenho da execução dos algoritmos quânticos se comparado com a execução em *CPUs multi-core*.

Agradecimentos

Agradecimento especial à Prof. Dra. Andrea Schwertner Charão e à Prof. Dra. Juliana Kaizer Vizzotto pelas contribuições no desenvolvimento deste trabalho.

Referências

Belkner, P. (2012). Eqcs home page. http://home.snafu.de/pbelkner/eqcs/.

Butscher, B. and Weimer, H. (2013). Libquantum - simulation of quantum mechanics. http://www.libquantum.de/.

Feitosa, S. S., Vizzotto, J. K., Piveta, E. K., and Du Bois, A. R. (2015). FJQuantum: uma Linguagem Quântica Orientada a Objetos. In *3nd Workshop-School on Theoretical Computer Science, WEIT 2015, Porto Alegre, RS, Brazil, October 14-16, 2015*, pages 136–143.

Gheorghiu, V. (2016). Quantum++ home page. http://vsoftco.github.io/qpp/.

Google (2016). Quantum computing playground. http://qcplayground.withgoogle.com.

Gruska, J. (2000). *Quantum Computing*. Mcgraw Hill Book.

Microsoft (2016). Liqui: The language integrated quantum operations simulator. http://stationq.github.io/Liquid/.

Nielsen, M. A. and Chuang, I. L. (2011). *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition.

NVidia (2016). Cuda toolkit documentation. http://docs.nvidia.com/cuda.

Ömer, B. (1998). A procedural formalism for quantum computing. Technical report, Technical University of Vienna.

OpenAcc (2016). Openacc home. http://www.openacc.org/.

Sanders, J. W. and Zuliani, P. (1999). Quantum programming. In *In Mathematics of Program Construction*, pages 80–99. Springer-Verlag.

Selinger, P. (2008). Finite dimensional hilbert spaces are complete for dagger compact closed categories. In *Proceedings of the 5th International Workshop on Quantum Physics and Logic (QPL 2008)*, page 11 pages, Reykjavik, Iceland.

van Tonder, A. (2004). A Lambda calculus for quantum computation. *SIAM Journal on Computing*, 33:1109–1135.

Vizzotto, J. K., Calegaro, B. C., and Piveta, E. K. (2013). A double effect λ-calculus for quantum computation. In Du Bois, A. R. and Trinder, P., editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg.

Williams, C. P. (2008). *Explorations in Quantum Computing*. Springer Publishing Company, Incorporated, 2nd edition.

Yanofsky, N. S. and Mannucci, M. A. (2008). *Quantum Computing for Computer Scientists*. Cambridge University Press.