Extensão da Biblioteca de Execução do Ambiente VPE-qGM para Suporte à Simulação Quântica em C++*

Murilo F. Schmalfuss[1], Adriano K. Maron[1], Renata H. S. Reiser[1], Maurício L. Pilla[1]

[1]Centro de Desenvolvimento Tecnológico – Universidade Federal de Pelotas (UFPel)
Caixa Postal 354 – 96.001-970 – Pelotas – RS – Brazil

Resumo. *As duas principais contribuições deste trabalho são o desenvolvimento da qGM$_C$-Analyzer, uma extensão da qGM-Analyzer implementada em C++, e sua integração com o ambiente de programação visual VPE-qGM para suporte futuro à simulações quânticas via OpenMP e GPUs. A ánalise e validação dos resultados são executadas por operações unitárias e controladas de 5 a 17 qubits.*

1. Introdução

A Computação Quântica (CQ) está fundamentada nos postulados definidos pela Mecânica Quântica (MQ), provendo a fundamentação e a concepção de algoritmos quânticos. Em vários cenários esses algoritmos são exponencialmente mais rápidos que seus análogos clássicos [Grover 1996, Shor 1997]. Dentre os fenômenos quânticos explorados pela CQ, destaca-se o **paralelismo quântico**, o qual está diretamente associado ao ganho de desempenho esperado. Entretanto, tais algoritmos exploram esses fenômenos quando executados em um *hardware* quântico, o qual está em fase de desenvolvimento e atualmente não suporta sistemas com muitos *qubits*.

O ambiente *VPE-qGM* (*Visual Programming Environment for the Quantum Geometric Machine Model*)[Maron et al. 2010] está em desenvolvimento com o objetivo de auxiliar na modelagem e simulação, sequencial e distribuída de algoritmos quânticos, apresentando as construções e a evolução dos estados dos sistemas quânticos a partir de um conjunto de *interfaces* gráficas desenvolvidas em *Python*. Considerando o alto custo computacional inerente à simulação de algoritmos quânticos em computadores clássicos, propõe-se a extensão das capacidades de simulação do *VPE-qGM* com suporte à integração de bibliotecas para processamento paralelo (*OpenMP* [Ayguade and Chapman 2003], *CUDA* [NVIDIA 2009]).

Neste sentido, a principal contribuição deste trabalho consiste no **desenvolvimento da biblioteca de execução *qGM$_C$-Analyzer***, uma extensão da *qGM-Analyzer* na linguagem *C++*, e sua integração ao ambiente *VPE-qGM* para a simulação paralela de algoritmos quânticos. A interface entre as duas linguagens de programação é obtida através da biblioteca *Boost* [BOOST 2012]. Os esforços relacionados a este trabalho se justificam pelo melhor desempenho da linguagem *C++* frente à *Python*, além de viabilizar o uso de bibliotecas de processamento paralelo, amplamente utilizadas pela comunidade científica.

2. Fundamentação

O ambiente *VPE-qGM*, fundamentado no modelo de processos *qGM* (*Quantum Geometric Machine Model*) [Reiser and Amaral 2010], é constituído de construtores para

*Edital CNPq Processo 128697/2012-1 Edital PIBIC/CNPq 2012/2013

composição, sincronização e soma não-determinística que são aplicados aos componentes gráficos que descrevem as aplicações. A simulação se dá a partir da modelagem obtida a partir desses construtores, apresentando os resultados a partir de interfaces gráficas. De acordo com o modelo *qGM*, a noção de portas quânticas pode ser substituída pelo conceito de sincronização de processos elementares (*PEs*). No ambiente *VPE-qGM*, o *PE* é um elemento estruturado por três atributos: (i) *Ação*: corresponde às transformações quânticas aplicadas a diferentes *qubits* em um mesmo instante de tempo; (ii) *Parâmetros*: contém dados auxiliares associados à definição das transformações quânticas; (iii) *Posição*: na qual é armazenado o resultado calculado pelo *PE*, no espaço de memória global e compartilhada.

Neste contexto, uma transformação quântica, aplicada à *N qubits*, pode ser modelada pela sincronização de 2^N *PEs*, cujas parametrizações satisfazem as condições equivalentes à definição dos vetores componentes da matriz (transformação unitária ou de medida) associada. Assim, durante a simulação, ocorre a execução (sequencial ou síncrona) dos *PEs*, os quais têm suas correspondentes computações efetuadas pela biblioteca *qGM-Analyzer*, manipulando os dados presentes nas posições de memória e simulando o comportamento de um sistema quântico.

2.1. Biblioteca qGM-Analyzer

A biblioteca de execução dos *PEs*, denominada *qGM-Analyzer*, implementa otimizações que controlam o aumento exponencial dos vetores componentes das Matrizes de Definição do Operador ($MDOs$) de múltiplos *qubits*, conforme introduzido em [Maron et al. 2011]. Os resultados relacionados comprovam a redução no consumo de memória durante a simulação, suportando algoritmos com 11 *qubits*. Entretanto, o tempo total de simulação obtido permanece elevado. Tal desempenho está fortemente relacionado com a linguagem de programação adotada pelo ambiente *VPE-qGM*. Por ser uma linguagem interpretada, *Python* apresenta desempenho inferior às linguagens tradicionalmente utilizadas para solução de problemas computacionalmente intensivos. Nesse sentido, justifica-se a extensão da biblioteca *qGM-Analyzer*, considerando a linguagem *C++*.

3. Extensão da qGM-Analyzer em C++

A implementação da biblioteca *qGM$_C$-Analyzer*, além de incluir as otimizações já introduzidas em [Maron et al. 2011], também altera as estruturas de dados fazendo uso de recursos nativos oferecidos pela linguagem, visando a otimização da execução. Para compatibilidade da biblioteca com o ambiente *VPE-qGM*, considera-se a biblioteca *Boost 1.49.0*, integrando as interfaces entre *Python* e *C++*.

A biblioteca *Boost* oferece diversos recursos e ferramentas que estendem a *STL* (*Standard Template Library*), oferecida pela linguagem *C++*. Desses recursos, exploram-se, especificamente, as funcionalidades providas pelo módulo *Boost-Python*. Usando funções definidas na biblioteca *Boost* é possível especificar quais métodos tornar-se-ão visíveis para o interpretador *Python*.

A ferramenta *Boost-Jam* auxilia no processo de compilação e integração da biblioteca, provê a leitura do código-fonte e gera a biblioteca compartilhada utilizando o compilador *GCC* 4.6.3 com as seguintes *flags* de compilação: *-O3* (identifica o nível de otimização realizada pelo compilador) e *-funroll-loops* (desenrolamento de laços). Esta

biblioteca compartilhada pode ser então importada e utilizada dentro do ambiente. A biblioteca *Boost-Python* se encarrega de executar as conversões dos parâmetros dos algoritmos e aplicações desenvolvidas.

A nova abordagem da *qGM$_C$-Analyzer* apresenta duas importantes otimizações. Primeiramente, sabe-se que a linguagem *Python* não oferece uma estrutura de dados de armazenamento contíguo, sendo a utilização de listas a alternativa frequentemente adotada. Entretanto, tal opção ocasiona um custo de indexação $O(n)$ [Cormen et al. 2000], onde n indica o número de elementos da lista. Neste contexto, a implementação em *Python* da biblioteca *qGM-Analyzer* faz uso de diversas listas, sendo estas substituídas, na nova abordagem, por vetores que possuem complexidade de indexação $O(1)$.

A outra modificação realizada para reduzir a complexidade foi a passagem dos parâmetros por referência para as funções auxiliares, eliminando assim o *overhead* da cópia. Como *Python* possui tipagem dinâmica, a resolução dos tipos das variáveis era feita em tempo de execução. Na nova abordagem com todas as variáveis utilizadas sendo declaradas no código-fonte, essa resolução é feita em tempo de compilação, melhorando o desempenho e permitindo ao compilador otimizações no código gerado.

4. Resultados

Os estudos de casos para validação e análise de desempenho da implementação da *qGM$_C$-Analyzer* em *C++* contemplam sistemas entre 5 e 17 *qubits*. A metodologia dos testes considera, para cada estudo de caso, a realização de 10 simulações. A máquina utilizada possui as seguintes características: processador Intel Core i7-3770 @ 3.5 GHz, 8GB RAM e sistema operacional Ubuntu 12.04 64 *bits*. Foram monitorados o tempo de execução de cada amostra e o correspondente consumo de memória. A principal comparação de desempenho se dá com a biblioteca *qGM-Analyzer* implementada em *Python*.

Tabela 1. Tempos de simulação

Operação	Qubits	Python		C++	
		Tempo (s)	Mem (MB)	Tempo (s)	Mem (MB)
$Hadamard$ 6 (H 6)	6	0,058	13	0,008	15
$Hadamard$ 7 (H 7)	7	0,0708	13	0,011	15
$Hadamard$ 8 (H 8)	8	0,182	13	0,0314	15
$Hadamard$ 9 (H 9)	9	0,556	13	0,096	15
$Hadamard$ 10 (H 10)	10	1,898	13	0,329	15
$Hadamard$ 11 (H 11)	11	8,170	13	1,235	15
$Hadamard$ 12 (H 12)	12	24,904	13	4,436	15
$Hadamard$ 13 (H 13)	13	87,040	13	16,307	15
$Hadamard$ 14 (H 14)	14	324,506	14	62,807	16
$Hadamard$ 15 (H 15)	15	1238,691	14	244,753	20
$Hadamard$ 16 (H 16)	16	7125,83	16	1062,291	24
$Hadamard$ 17 (H 17)	17	24138,29	18	3944,302	29
$Pauli\ X$ (X)	17	61,450	18	15,349	29
Controladas 7 (Cont. 7)	7	0,026	13	0,006	15
Controladas 5 (Cont. 5)	5	0,006	13	0,001	15
Controladas 14 (Cont. 14)	14	7,253	14	2,063	16
Controladas 16 (Cont. 16)	16	210,462	16	37,284	22

Como esperado, o aumento exponencial na quantidade de operações para os estudos de caso baseados em operações *Hadamard* gera elevado tempo de simulação. Para as

transformações *Pauli X*, o tempo de execução foi menor do que os estudos de caso com operações *Hadamard*, apesar da maior quantidade de *qubits* aplicada. Tal comportamento se refere a redução dos valores gerados por *PEs*.

Devido à arbitrariedade das configurações aplicadas, as operações compostas por transformações quânticas controladas apresentam dois comportamentos: (i) o primeiro, resultando em um baixo tempo de execução, sem aplicação de transformações *Hadamard*; (ii) o segundo, resultando em tempo de simulação maior devido a presença de várias transformações *Hadamard*.

5. Conclusões

As otimizações realizadas na *qGM$_C$-Analyzer* apresentaram bom desempenho em relação à biblioteca anterior, reduzindo significativamente o tempo de simulação dos *PEs*. Essa redução no tempo permitirá incremento no desempenho referente a simulação paralela/distribuída de sistemas quânticos de multiplos *qubits*.

A análise da eficiência da *qGM$_C$-Analyzer* para determinadas configurações de transformações quânticas acima 17 *qubits* passa a ser desafio, uma vez que o tempo de simulação cresce exponencialmente com o aumento na quantidade de *qubits*.

A continuidade do trabalho consiste na extensão da biblioteca para execução paralela, integrando com módulos e bibliotecas para paralelização disponíveis para *C++* como *OpenMP*, e também para paralelização massiva utilizando placas gráficas através da arquitetura *CUDA*. A *API OpenMP* é direcionada para o uso com *C++*, possibilitando sua utilização futura na biblioteca. Estudos já estão sendo realizados para a paralelização da biblioteca.

Referências

Ayguade, E. and Chapman, B. (2003). Introduction: Special issue: OpenMP. *Scientific Programming*, 11(2):79–80.

BOOST (2012). Boost 1.49.0 library documentation. www.boost.org/doc/libs/1_49_0/.

Cormen, T., Leiserson, C., and Rivest, R. (2000). *Introduction to Algorithms*. McGraw-Hill.

Grover, L. (1996). A fast quantum mechanical algorithm for database search. *Proc. of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 212–219.

Maron, A., Ávila, A., Reiser, R., and Pilla, M. (2011). Introduzindo uma nova abordagem para simulação quântica com baixa complexidade espacial. In *Anais do DINCON 2011*, pages 1–6. SBMAC.

Maron, A., Pinheiro, A., Reiser, R., Yamin, A., and Pilla, M. (2010). Ambiente VPE-qGM: Em direção a uma nova abordagem para simulações quânticas. *Rev. do CCEI*, 14:29–46.

NVIDIA (2009). *CUDA Programming Guide*. NVIDIA Corp. Version 2.3.

Reiser, R. and Amaral, R. (2010). The quantum states space in the qgm model. In *Anais/III WECIQ*, pages 92–101, Petrópolis/RJ. Editora do LNCC.

Shor, P. (1997). Polynomial-time algoritms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*.