UNIVERSIDADE FEDERAL DE PELOTAS

Centro de Desenvolvimento Tecnológico

Programa de Pós-Graduação em Computação

Dissertação

Explorando as Possibilidades de Otimização da Simulação de Algoritmos Quânticos no VPE-qGM

ADRIANO KURZ MARON

Pelotas, 2013

ADRIANO KURZ MARON

Explorando as Possibilidades de Otimização da Simulação de Algoritmos Quânticos no VPE-qGM

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Profa. Dra. Renata Hax Sander Reiser
Co-orientador: Prof. Dr. Maurício Lima Pilla

Pelotas, 2013

Dados Internacionais de Publicação (CIP)

M354e Maron, Adriano Kurz
Explorando as possibilidades de otimização da simulação de algoritmos quânticos no VPE-qGM / Adriano Kurz Maron; Renata Hax Sander Reiser, orientador; Maurício Lima Pilla, co-orientador. – Pelotas, 2013.
82 f.; il.

Dissertação (Mestrado em Ciência da Computação), Centro de Desenvolvimeno Tecnológico/Programa de Pós Graduação em Computação, Universidade Federal de Pelotas. Pelotas, 2013.

1. Simulação quântica aaralela. 2. Programação em GPU. 3. Processos quânticos. I. Reiser, Renata Hax Sander orient. II. Pilla, Maurício Lima co-orient. III. Título.

CDD: 511.8

Catalogação na Fonte: Leda Cristina Peres Lopes CRB:10/2064
Universidade Federal de Pelotas

Banca examinadora:

Profª. Drª. Juliana Kaizer Vizzotto

Profª. Drª. Simone Cavalheiro

Prof. Dr. André Rauber Du Bois

AGRADECIMENTOS

Aos meus pais, Gilberto e Mariseti, por me apoiarem durante mais essa etapa de estudos e evolução pessoal, contribuindo diretamente para a conquista de mais esse objetivo.

Ao meu irmão, Guilherme, por proporcionar bons momentos de distração e diversão.

À professora Renata Reiser que, além de me acompanhar durante todo meu curso de graduação, acreditou no meu potencial e manteve essa parceria neste curso de Mestrado, me ajudando de várias formas para que pudéssemos desenvolver um trabalho de qualidade e, mais importante, para que pudéssemos manter essa amizade que já se estende por quase 7 anos.

Aos professores Maurício Pilla e Adenauer Yamin que sempre contribuem com críticas e sugestões essenciais para o andamento deste projeto.

Aos meus colegas de Mestrado que me proporcionam muitos momentos divertidos.

À todos os meus amigos que estão sempre dispostos a me ajudar de várias formas e em vários momentos distintos.

Obrigado a todos.

O sábio nunca diz tudo o que pensa, mas pensa sempre tudo o que diz. —Aristóteles

RESUMO

MARON, Adriano Kurz. **Explorando as Possibilidades de Otimização da Simulação de Algoritmos Quânticos no VPE-qGM**. 2013. 82 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

A simulação de algoritmos quânticos a partir de computadores clássicos consiste em uma metodologia de estudo, desenvolvimento e validação que busca complementar a abordagem teórica aplicada nos estágios iniciais de construção dos algoritmos em questão. Entretanto, tal simulação é caracterizada por um elevado custo de processamento e armazenamento, exigindo recursos computacionais em larga escala. Visando a obtenção de soluções para uma simulação mais eficiente, este trabalho propõe uma metodologia de desenvolvimento caracterizada por duas etapas principais: (i) estudo teórico e implementação sequencial das abstrações de Processos Quânticos e Processos Quânticos Parciais definidos no modelo *qGM*, visando a redução no consumo de memória associado à transformações quânticas multidimensionais; (ii) implementação paralela dessas abstrações para correspondente execução sobre a arquitetura massivamente paralela das *GPUs*. Os resultados obtidos neste trabalho contemplam a simulação sequencial de transformações quânticas controladas de até 24 *qubits*. No âmbito da simulação paralela, transformações *Hadamard* de até 20 *qubits* foram simuladas com *speedup* de $\approx 185\times$ sobre uma simulação distribuída com 8 *cores*, caracterizando uma excelente melhora de desempenho no ambiente *VPE-qGM* com relação a suas limitações anteriores. Este trabalho estabelece as diretrizes para o desenvolvimento de extensões da biblioteca de simulação do ambiente e das capacidades de simulação paralela para transformações quânticas controladas e operações de medida.

Palavras-chave: Simulação Quântica Paralela, Programação em GPU, Processos Quânticos.

ABSTRACT

MARON, Adriano Kurz. **Exploring the Optimization Possibilities for the Simulation of Quantum Algorithms in the VPE-qGM**. 2013. 82 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

The simulation of quantum algorithms using classical computers consists in a methodology for the study, development and validation that aims to complement the theoretical approach applied in the initial stages of the creation of the aforementioned algorithms. However, such simulation is characterized by a high processing and storage costs, often requiring large scale computational resources. In order to provide novel solutions for a more efficient simulation, this work proposes a development methodology defined by two main steps: (i) the first comprehends the theoretical studies and sequential implementation of the abstractions corresponding to the Quantum Processes and Quantum Partial Processes defined in the qGM model, focusing on the reduction of the memory consumption regarding multidimensional quantum transformations; (ii) the second considers the parallel implementation of such abstractions allowing its execution the massive parallel architecture of the *GPUs*. The results obtained by this work embrace the sequential simulation of controlled transformations up to 24 qubits. In the parallel simulation approach, *Hadamard* gates up to 20 qubits were simulated with a speedup of $\approx 185\times$ over a 8-core distributed simulation, being a significant performance improvement in the *VPE-qGM* environment when comparing with its previous limitations. This work establishes the directions for the next steps of the development of the simulation library of the environment, allowing the extension of the parallel simulation capabilities for controlled quantum gates and measurement operations.

Keywords: Parallel Quantum Simulation, VPE-qGM Environment, GPU Programming, Quantum Process.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

VPE-qGM	*Visual Programing Environment for the Quantum Geometric Machine Model*
qGM	*Quantum Geometric Machine*
qPE	*Quantum Process Editor*
qME	*Quantum Memory Editor*
qS	*Quantum Simulator*
PS	Produto Sequencial
PP	Produto Paralelo
VirD-GM	*Virtual Distributed Geometric Machine*
CQ	Computação Quântica
MQ	Mecânica Quântica
PE	Processo Elementar
QP	*Quantum Process*
QPP	*Quantum Partial Process*
QuIDD	*Quantum Information Decision Diagram*
MPQCS	*Massive Parallel Quantum Computer Simulator*
GPU	*Graphic Processing Unit*
CUDA	*Compute Unified Device Architecture*

SUMÁRIO

1 INTRODUÇÃO

Este trabalho está inserido no Projeto *ExPloreD-GM - Explorando o Paralelismo e a Distribuição do Modelo D-GM em Aplicações Científicas e Tecnologias Associadas*, buscando o estudo e aplicação de otimizações para simulação, sequencial e paralela, de algoritmos quânticos a partir de computadores clássicos. Em trabalhos anteriores, os quais contemplaram a especificação e implementação do ambiente *VPE-qGM* (*Visual Programming Environment for the Quantum Geometric Machine Model*) (MARON et al., 2010, 2011), viabilizou-se a modelagem e simulação distribuída de algoritmos quânticos, apresentando as construções e evolução dos sistemas quânticos a partir de um conjunto de *interfaces* gráficas.

Essa abordagem visa simplificar o estudo dos conceitos associados a computação quântica, contribuindo para o desenvolvimento prático de algoritmos quânticos. Entretanto, devido ao alto custo computacional decorrente da simulação de algoritmos quânticos complexos, o *VPE-qGM* fica limitado a sistemas de aproximadamente 14 *bits* quânticos ($qubits$), o que resulta em, por exemplo, um algoritmo quântico para busca em listas de 2^{13} elementos.

1.1 Motivação

A Computação Quântica (CQ) é um paradigma fundamentado nos postulados definidos pela Mecânica Quântica (MQ), a qual provê interpretações para os comportamentos físicos incomuns que se fazem presentes quando da manipulação de elementos em escala atômica/subatômica, substituindo, dessa forma, as leis da Física Clássica. A aplicação dessas propriedades no contexto computacional permite a concepção de uma nova classe de computadores (computadores quânticos), os quais são capazes de apresentar um desempenho exponencialmente maior do que os computadores clássicos.

O estado-da-arte em *hardware* quântico ainda apresenta baixa escalabilidade devido a dificuldade de manipulação e controle de partículas elementares (elétrons, fótons, etc), as quais têm potencial para serem utilizadas como *qubits*. Assim, atualmente, o estudo e desenvolvimento de algoritmos para *CQ* pode ser realizado de duas formas principais:

através da especificação matemática do sistema ou do desenvolvimento dos circuitos quânticos em *softwares* de simulação.

Contudo, a simulação de sistemas quânticos através de computadores clássicos mostra-se ineficiente, visto o alto custo computacional associado à aplicação das transformações que determinam a evolução temporal do sistema. Nesse cenário, a adoção de soluções voltadas para o ganho de desempenho é essencial para simulação de aplicações quânticas complexas.

O estudo e implementação, no ambiente *VPE-qGM*, de técnicas para manipulação e realização de computações a partir de uma grande quantidade de dados provê suporte à uma simulação mais eficiente de algoritmos quânticos. Essas técnicas, aliadas à alta capacidade computacional de algumas arquiteturas multicomputadores/multiprocessadas (*clusters* e GPUs, respectivamente) têm potencial para suportar a simulação de algoritmos quânticos com muitos *qubits*.

1.2 Objetivos

Este trabalho busca estudar novas técnicas de paralelização e otimização das estruturas associadas a transformações quânticas, voltados para simulação via *software* de algoritmos da *CQ*. A partir desse estudo tem-se a possibilidade de aplicação de soluções eficientes no ambiente *VPE-qGM*, visando a extensão das suas capacidade de simulação. Mais especificamente, os seguintes tópicos principais são contemplados por este trabalho:

- o estudo e implementação de abstrações presentes no modelo *qGM* para interpretação de transformações quânticas a partir de Processos Quânticos e Processos Quânticos Parciais, sendo uma abordagem diferenciada com relação aos demais trabalhos da área;
- o desenvolvimento da extensão para suporte a simulação quântica paralela a partir de *GPUs*.

Esses esforços tem por objetivo consolidar o *VPE-qGM* como um ambiente de simulação com sólida fundamentação matemática e alta capacidade computacional pela exploração de *clusters* e *GPUs*, utilizados para distribuição e paralelização das computações associadas a evolução dos sistemas quânticos.

1.3 Metodologia

A metodologia que caracteriza a consolidação deste trabalho considera, em primeira instância, o estudo de abstrações contidas no modelo *qGM* para definição das estratégias de otimização aplicáveis no ambiente *VPE-qGM*. Tem-se, como principal meta

desse estudo, a implementação dessas otimizações que, além de expandir as capacidades de simulação do *VPE-qGM*, consolidam as possibilidades de interpretação do modelo *qGM*, explicitando sua expressividade quando da descrição de transformações/algoritmos quânticos.

Após a consolidação e validação das otimizações para representação de transformações quânticas, tem-se o estudo e prototipação da extensão para simulação quântica paralela a partir de *GPUs*. A busca por uma execução eficiente das computações associadas às **transformações quânticas básicas** caracteriza a principal contribuição desta etapa, definindo as premissas a serem seguidas quando da extensão destas implementações para suporte as demais configurações de transformações quânticas (transformações controladas, personalizadas e medidas projetivas).

A obtenção de resultados reais neste trabalho se dá pela análise da simulação, sequencial e paralela, de transformações quânticas multi-qubit, validando as implementações realizadas. Essas tarefas projetam os futuros avanços relacionados a este trabalho, contribuindo para uma nova abordagem voltada a simulação de algoritmos quânticos em computadores clássicos.

1.4 Organização do Texto

Este trabalho está estruturado da seguinte forma: No Capítulo 2 os principais fundamentos relacionados à computação quântica são descritos, incluindo uma breve discussão, na Seção 2.2, acerca das características encontradas nos simuladores quânticos atualmente disponíveis. Um estudo que abrange os principais conceitos sobre processamento em *GPUs* é apresentado no Capítulo 3, com destaque para a Seção 3.5 que contém algumas das políticas de otimização exigidas para essa forma de processamento. A principal contribuição deste trabalho começa a ser descrita no Capítulo 4, abrangendo as etapas de estudo e implementação dos conceitos de Processos Quânticos e Processos Quânticos Parciais. Na sequência, o Capítulo 5 contempla as implementações direcionadas ao suporte à simulação paralela através de *GPUs*, detalhando a forma de estruturação dos Processos Quânticos e o algoritmo utilizado. No Capítulo 6, uma análise de desempenho das simulações sequenciais e paralelas é realizada de forma a identificar as novas limitações do ambiente *VPE-qGM*. Por fim, no Capítulo 7, as principais conclusões obtidas a partir da realização deste trabalho, bem como propostas de continuidade do projeto como um todo são apresentadas.

2 COMPUTAÇÃO QUÂNTICA

O corrente capítulo contempla os principais conceitos de *CQ* e *MQ* que fundamentam este trabalho. Também são estudados alguns dos simuladores quânticos disponíveis atualmente.

2.1 Conceitos Básicos

A manipulação de partículas em escala atômica/subatômica compreende uma tarefa de alta complexidade, visto que nessas situações as partículas (fótons, elétrons e outras partículas de mesma escala) exibem comportamentos incomuns, não definidos pelas leis da física clássica. A *MQ* é a área da física que estuda tais comportamentos, apresentando teorias que definem de forma precisa os fenômenos que ocorrem em pequena escala.

Um dos principais fundamentos da *MQ* é a ocorrência de **superposição de estados** (PESSOA, 2003). Na mecânica clássica, é estabelecido que uma partícula possui um estado único e bem definido, como por exemplo, um elétron com um *spin* positivo. Na *MQ*, uma partícula pode ser definida a partir da coexistência de dois ou mais estados, ou seja, um elétron com *spin* positivo e negativo simultaneamente.

A *MQ* também contempla a interpretação do fenômeno da **dualidade onda-partícula**, exemplificado através do Experimento da Fenda Dupla (THE DOUBLE-SLIT EXPERIMENT, 2009), na qual uma partícula atômica é capaz de apresentar dois comportamentos distintos: (i) ondulatório, onde sua trajetória é descrita por uma superposição de ondas; (ii) corpuscular, com uma trajetória bem definida.

Uma partícula quântica é definida através de uma **função de onda**, sendo constituída de uma grande variedade de estados possíveis. Porém, ao utilizar qualquer dispositivo para medir (observar) o estado dessa partícula, sua função de onda colapsa em uma das possibilidades, comportando-se, a partir desse ponto, como uma partícula com um estado bem definido.

Desse comportamento surge o **princípio da incerteza de Heisenberg** (PESSOA, 2003) que, em suma, estabelece que ao medir o estado de um objeto quântico, instantaneamente seu estado será alterado, deixando de apresentar propriedades quânticas.

Nota-se, portanto, a impossibilidade de determinar a trajetória de uma partícula quântica, visto que, ao medir a correspondente posição, seu estado colapsa, impedindo a medida de sua velocidade.

O princípio do **entrelaçamento** (PESSOA, 2003) considera que duas partículas criadas juntas são capazes de interagir de forma imediata mesmo quando separadas espacialmente por qualquer distância, de forma que, ao submeter uma partícula a um determinado efeito, a outra partícula entrelaçada a esta irá reagir instantaneamente. Interpretações detalhadas dos princípios da *MQ* podem ser obtidas em (PESSOA, 2003; PORTUGAL; LAVOR; MACULAN, 2004).

Essas ponderações iniciais fundamentam a descrição do comportamento de sistemas quânticos, os quais são matematicamente especificados através de quatro postulados definidos pela *MQ*, permitindo a analogia com sistemas físicos.

Essas ponderações iniciais fundamentam a descrição do comportamento de sistemas quânticos, os quais são matematicamente especificados através de quatro postulados definidos pela *MQ*, permitindo a analogia com sistemas físicos. Descrições resumidas desses postulados são apresentados nas seções seguintes.

2.1.1 Primeiro Postulado: Espaço de Estados

O primeiro postulado define a forma de representação de um sistema quântico, descrito por construções fundamentadas na álgebra linear. De acordo com (NIELSEN; CHUANG, 2003), qualquer sistema físico isolado está associado a um espaço vetorial complexo com um produto interno (ou seja, um espaço de Hilbert), conhecido como espaço de estados do sistema. O sistema é completamente descrito pelo seu vetor de estado, um vetor unitário no espaço de estados.

O sistema quântico mais simples é o *qubit* (*quantum bit*), o qual é definido por um espaço de estados bidimensional, descrito por um estado genérico que, na notação de Dirac, é representado pela expressão

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{1}$$

onde α e β são coeficientes complexos associados às amplitudes dos estados $|0\rangle$ e $|1\rangle$, respectivamente. Desta definição, destaca-se que $|0\rangle$ e $|1\rangle$ possuem, respectivamente, a seguinte representação através da notação matricial:

$$\begin{pmatrix}1\\0\end{pmatrix} \; e \; \begin{pmatrix}0\\1\end{pmatrix} \tag{2}$$

Analogamente, $|\psi\rangle$ possui a seguinte representação como uma matriz coluna:

$$\begin{pmatrix}\alpha\\\beta\end{pmatrix}. \tag{3}$$

A definição do sistema deve corresponder a uma **base ortonormal** para o espaço de estados. Considerando as definições expressas em 2 e 3 para o *qubit* $|\psi\rangle$, a base ortonormal deve obedecer as seguintes propriedades:

- Ortogonalidade dos Vetores de Estados: O produto interno entre os vetores correspondentes aos estados do sistema deve ser igual a 0. Dessa forma,

$$\langle 0|1\rangle = 0 \;\equiv\; (1,0)\begin{pmatrix}0\\1\end{pmatrix} = 0. \tag{4}$$

- Normalização das Probabilidades dos Estados: O vetor de estado do sistema deve satisfazer a condição de unitariedade, sendo equivalente a $|\alpha|^2 + |\beta|^2 = 1$. A **condição de normalização** do vetor de estado pode ser representada por $\langle\psi|\psi\rangle = 1$, sendo escrita como

$$(\alpha,\beta)\begin{pmatrix}\alpha\\\beta\end{pmatrix} = \alpha^2 + \beta^2 = 1. \tag{5}$$

2.1.2 Segundo Postulado: Evolução

Em (NIELSEN; CHUANG, 2003), está descrito que a evolução de um sistema quântico isolado se dá através de transformações unitárias. Ou seja, o estado $|\psi^1\rangle$ do sistema no tempo t_1 está relacionado com o estado $|\psi^2\rangle$ do sistema no tempo t_2 através de um operador unitário U que depende apenas de t_1 e t_2:

$$|\psi^2\rangle = U|\psi^1\rangle \tag{6}$$

Por esse postulado, entende-se que, para realizar qualquer alteração sobre o estado de um sistema quântico, é necessário que o operador de transformação preserve a característica de normalização do vetor de estado. Dessa forma, uma transformação quântica deve ser **unitária**, satisfazendo a condição $U^tU = Id$.

2.1.3 Terceiro Postulado: Medidas Quânticas

As medidas quânticas estão relacionadas com o processo de observação do estado de um sistema quântico. Em uma descrição formal, obtida em (NIELSEN; CHUANG, 2003), as medidas quânticas são descritas por determinados operadores de medida M_m, sendo $1 < m < 2^n - 1$, para sistemas de *n qubits*. Estes operadores atuam sobre o espaço de estados do sistema. O índice *m* se refere aos possíveis resultados da medida. Se o estado do sistema quântico for $|\psi\rangle$ imediatamente antes da medida, a probabilidade de um resultado *m* é dada por

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^{\dagger 1}M_m|\psi\rangle}}, \tag{7}$$

satisfazendo a relação de completude

$$\sum_m M_m^\dagger M_m = I. \tag{8}$$

O **conjunto básico** de operadores de medida é compreendido por:

$$\left\{ M_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} M_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right\}. \tag{9}$$

Os operadores de medida não são reversíveis, ou seja, uma vez efetuada a medida sobre um estado quântico, o sistema instantaneamente colapsa em um de seus estados clássicos. Considerando a medida sobre o *qubit* $|\psi\rangle$, o estado final da sistema pode ser $|0\rangle$ ou $|1\rangle$, com probabilidades definidas por:

$$\begin{aligned} p(|0\rangle) &= \langle\psi|M_0^\dagger M_0|\psi\rangle = \langle\psi|M_0|\psi\rangle = |\alpha|^2 \\ p(|1\rangle) &= \langle\psi|M_1^\dagger M_1|\psi\rangle = \langle\psi|M_1|\psi\rangle = |\beta|^2 \end{aligned} \tag{10}$$

2.1.4 Quarto Postulado: Sistemas Compostos

(NIELSEN; CHUANG, 2003) descreve que o espaço de estados de um sistema físico composto é o produto tensorial dos espaços de estados dos sistemas físicos individuais. Se os sistemas forem enumerados de 1 até n e o sistema i for preparado no estado $|\psi\rangle$, decorre que o estado do sistema composto será

$$|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle. \tag{11}$$

Considerando um sistema quântico de dois *qubits*, ψ e φ, os quais são definidos pelas expressões

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \; e \; |\varphi\rangle = \gamma|0\rangle + \delta|1\rangle, \tag{12}$$

o espaço de estados do sistema é composto pelo produto tensor ($|\psi\rangle \otimes |\varphi\rangle$), ou seja:

$$|\psi\varphi\rangle = \alpha \cdot \gamma|00\rangle + \alpha \cdot \delta|01\rangle + \beta \cdot \gamma|10\rangle + \beta \cdot \delta|11\rangle. \tag{13}$$

2.1.5 Computação Quântica como um Paradigma Computacional

A aplicação das propriedades da *MQ* no contexto computacional resulta no paradigma da *CQ*, permitindo o desenvolvimento de computadores quânticos com capacidade de processamento exponencialmente maior que os computadores clássicos, como demonstrado pela Tabela 1.

O modelo mais recorrente para descrição de aplicações quânticas é o modelo de

[1]Notação aplicada para a transposta conjugada a matriz M.

Tabela 1: Fatoração clássica x fatoração quântica

Tamanho do nº a ser fatorado	Algoritmo Clássico	Algoritmo Quântico
512 *bits*	4 dias	34 segundos
1024 *bits*	10^5 anos	4,5 minutos
2048 *bits*	10^{14} anos	36 minutos
4096 *bits*	10^{26} anos	4,8 horas

circuitos quânticos (NIELSEN; CHUANG, 2003). Essa representação é uma das mais fundamentais da *CQ*, sendo caracterizada por uma notação gráfica intuitiva que remete ao modelo de circuitos digitais utilizados na computação clássica.

Os circuitos quânticos compreendem sincronizações e composições de portas (transformações) quânticas unitárias e operações de medidas, modelando qualquer tipo de algoritmo quântico, conforme apresentado na Figura 1. Algumas convenções são adotadas visando uma descrição homogênea dos algoritmos quânticos, sendo descritas a seguir:

- Linhas Horizontais: Cada linha representa um dos *qubits* que compõem o circuito, e a correspondente evolução temporal ocorre da esquerda para a direita;
- Linhas Verticais: Indicam que uma determinada transformação quântica atua sobre os *qubits* conectados através desta linha;
- Controle: Representado por um círculo sobre a linha de um *qubit*. Se o círculo for fechado, indica que é considerado o estado $|1\rangle$ do *qubit*; se for aberto, considera-se o estado $|0\rangle$;
- Portas Quânticas: Transformações unitárias que manipulam o *qubit* sobre o qual são aplicadas;
- Medida: No final de cada linha do circuito pode aparecer uma operação de medida, determinando o estado clássico do correspondente *qubit*.

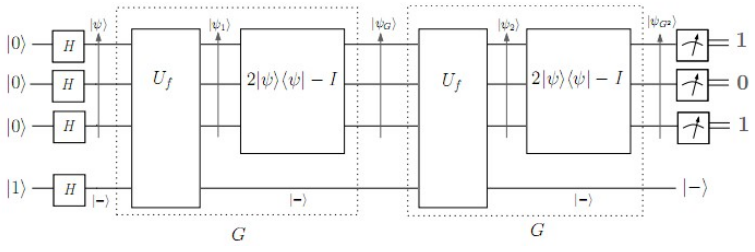


Figura 1: Circuito Quântico

2.1.6 Transformações Quânticas

As transformações unitárias quânticas são as operações responsáveis por manipular as amplitudes associadas aos estados do sistema. Essas transformações são definidas por matrizes unitárias quadradas de ordem 2^N, onde N representa a quantidade de *qubits* sobre os quais a transformação irá atuar. As principais transformações quânticas básicas são descritas na sequência.

- Hadamard: É a transformação responsável por gerar a superposição dos estados de um *qubit*. Sua definição matricial é:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{14}$$

  A aplicação de H sobre o vetor de estado do *qubit* genérico $|\psi\rangle$, definido na primeiro postulado da *MQ*, resulta em:

$$H|\psi\rangle = \frac{1}{\sqrt{2}}(\alpha + \beta, \alpha - \beta)^t \tag{15}$$

- Pauly X: Equivalente à porta clássica *NOT*, que inverte as amplitudes dos estados de um *qubit*. A correspondente definição matricial e aplicação sobre o vetor de estado de $|\psi\rangle$ é representado por:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} \tag{16}$$

- Pauly Y: Quando aplicada a $|\psi\rangle$, resulta em $Y|\psi\rangle = -i\beta|0\rangle + i\alpha|1\rangle$. A correspondente definição matricial é:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{17}$$

- Pauly Z: A matriz de transformação dessa operação é dada por:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{18}$$

  Sua função é realizar a inversão de fase do *qubit*, transformando o vetor de estado em $(\alpha, -\beta)^t$.

- Phase (S): Introduz uma fase relativa, ou seja, leva o *qubit* $|\psi\rangle$ para o estado $S|\psi\rangle = \alpha|0\rangle + i\beta|1\rangle$, onde a amplitude de $|0\rangle$ mantêm-se inalterada, enquanto que

a amplitude de $|1\rangle$ difere por um fator de fase igual à *i*. A matriz correspondente à porta Phase é descrita por:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{19}$$

- $\pi/8$: Transformação quântica associada a seguinte matriz unitária:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & exp(i\pi/4) \end{pmatrix} \tag{20}$$

A aplicação de T ao vetor de estado de $|\psi\rangle$ resulta em $(\alpha, exp(i\pi/4)\beta)^t$.

Para exemplificação do aumento exponencial nas transformações quânticas aplicadas a múltiplos *qubits*, considera-se, primeiramente, a transformação *Hadamard* (H) aplicada a um *qubit*. A seguinte representação matricial descreve tal cenário:

$$H|\psi\rangle \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} \alpha + \beta \\ \alpha - \beta \end{pmatrix} \tag{21}$$

Considerando, agora, a aplicação simultânea de H à dois *qubits* de um sistema quântico, tem-se a matriz

$$H^{\otimes 2} \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}, \tag{22}$$

a qual é obtida a partir da operação de produto tensorial $(\otimes)$ entre as correspondentes matrizes de 1 *qubit*. O operador $\otimes$ gera um aumento exponencial na quantidade de elementos da matriz resultante, influenciando significativamente no custo de simulação das operações.

A aplicação de $H^{\otimes 2}$ ao sistema quântico definido na Eq. 13, é dada por

$$\frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \times \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \alpha + \beta + \gamma + \delta \\ \alpha - \beta + \gamma - \delta \\ \alpha + \beta - \gamma - \delta \\ \alpha - \beta - \gamma + \delta \end{pmatrix} \tag{23}$$

Além das transformações de múltiplos *qubits*, obtidas pelo produto tensorial de transformações unitárias básicas, existem transformações controladas que modificam o estado de um ou mais *qubits* considerando o estado corrente dos demais. Dentre as transformações controladas, destacam-se:

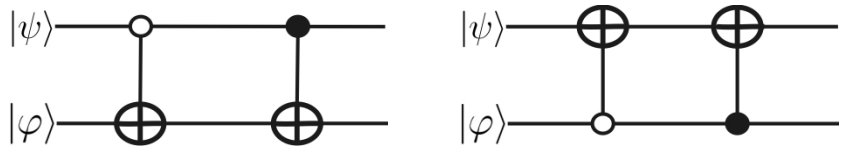
- *CNOT* (*Controled NOT*): A transformação quântica *CNOT* recebe 2 *qubits* $|\psi\rangle$ e $|\varphi\rangle$ como entrada e aplica a transformação *NOT* (*Pauly X*) a um deles (*qubit* alvo), considerando o estado corrente do outro *qubit* (controle). Duas configurações distintas para essa transformação são possíveis:

  – Controle no *Qubit* $|\psi\rangle$: A modificação no *qubit* $|\varphi\rangle$ pode ser realizada quando o estado de $|\psi\rangle$ for $|0\rangle$ ou $|1\rangle$. No modelo de circuitos quânticos, essas operações são apresentadas de acordo com a Figura 2(a). Assim, têm-se os seguintes operadores matriciais que manipulam o vetor de estado do sistema:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \\ \gamma \\ \delta \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{pmatrix} \tag{24}$$

  – Controle no *Qubit* $|\varphi\rangle$: A modificação no *qubit* $|\psi\rangle$ pode ser realizada quando o estado de $|\varphi\rangle$ for $|0\rangle$ ou $|1\rangle$. Analogamente ao caso anterior, as correspondentes representações no modelo de circuitos quânticos são apresentadas na Figura 2(b), respectivamente. Os dois operadores matriciais que definem o comportamento dessa transformação são:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \gamma \\ \beta \\ \alpha \\ \delta \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \\ \delta \\ \gamma \\ \beta \end{pmatrix} \tag{25}$$



(a) Portas com controle em $|\psi\rangle$ (b) Portas com controle em $|\varphi\rangle$

Figura 2: Transformações quânticas controladas CNOT

- *Toffoli*: A transformação controlada *Toffoli* é definida para três *qubits*, de forma a aplicar a transformação *Pauly X* ao terceiro *qubit* quando os estados dos dois primeiros *qubits* forem $|1\rangle$. A representação no modelo de circuitos quânticos é exemplificada pela Figura 3.

- *Controlled-U*: Na transformação controlada *CNOT*, o operador *Pauly X* é aplicado ao *qubit* alvo considerando o estado de um único *qubit* de controle. Porém, transformações controladas genéricas (*Controlled-U*) (NIELSEN; CHUANG, 2003)

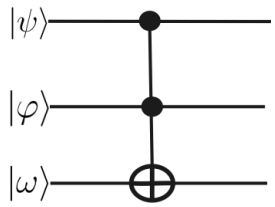


Figura 3: Porta *Toffoli* no modelo de circuitos quânticos.

podem ser definidas, de forma a utilizar variadas configurações de *qubits* de controle e aplicar qualquer transformação unitária U ao(s) *qubit(s)* alvo.

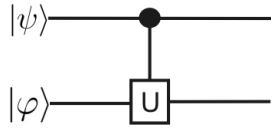


Figura 4: Porta *Controlled-U* no modelo de circuitos quânticos.

A partir da composição, sequencial e síncrona, de transformações quânticas, é possível realizar computações (GROVER, 1996; SHOR, 1997; KNILL; NIELSEN, 2000; AARONSON, 2007) que exploram os fenômenos particulares da mecânica quântica, obtendo ganhos frente aos melhores algoritmos clássicos conhecidos.

2.2 Simuladores Quânticos

Atualmente estão disponíveis simuladores quânticos com diversas abordagens. Dentre os mais relevantes, nesta seção são citados simuladores sequenciais que implementam otimizações para representação de transformações e de estados quânticos, reduzindo a quantidade de memória requerida durante a simulação, suportando sistemas quânticos com mais de 30 *qubits*. Também são consideradas abordagens para simulação paralela de algoritmos quânticos, focados na redução do tempo necessário para aplicação das transformações quânticas pela exploração de *clusters* e *GPUs*.

2.2.1 QuIDDPro

O simulador *QuIDDPro*, proposto em (VIAMONTES, 2007), utiliza estruturas denominadas *QuIDDs* (*Quantum Information Decision Diagrams*) para representar eficientemente transformações e estados quânticos multidimensionais, os quais são definidos, matricialmente, por blocos de valores repetidos. Esses padrões de repetição ocorrem com frequência em vários algoritmos, sendo possível obter uma significativa redução no consumo de memória e no tempo de acesso às informações.

Um *QuIDD* é uma representação comprimida de matrizes e vetores, permitido que computações sejam realizadas diretamente sobre essa estrutura otimizada. Exemplificando, na Figura 5, tem-se a representação de diferentes estados quânticos utilizado

QuIDDs. Na Figura 5(c), a aresta sólida saindo do vértice I_0 equivale a assinalar o valor 1 ao primeiro *bit* do índice de 2 *bits*. Já a aresta tracejada do vértice I_1 equivale a assinalar o valor 0 ao segundo *bit* do mesmo índice. Esses caminhos levam ao valor $-\frac{1}{2}$, o qual é a amplitude do estado da base computacional indexado por $|10\rangle$.

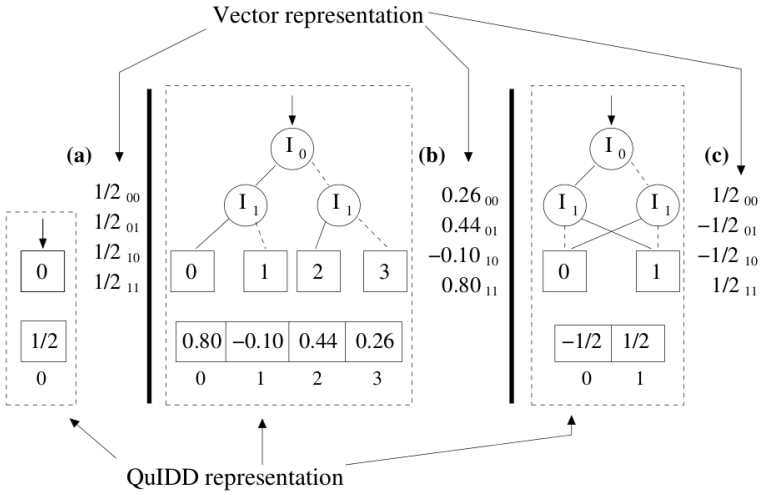
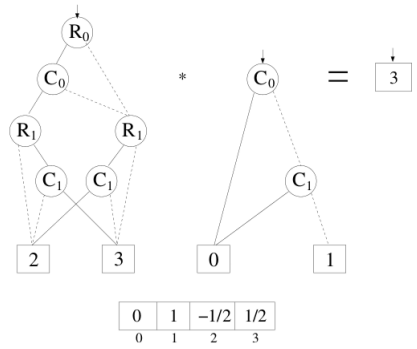


Figura 5: Exemplos de QuIDDs para diferentes estados quânticos (VIAMONTES, 2007).

Dessa representação percebe-se que, para estados com amplitudes iguais, é obtido um *QuIDD* extremamente simples. Para estados com muitas amplitudes diferentes, não é possível obter compressão. Entretanto, tais estados não são usuais na *CQ*. Transformações quânticas são definidas de forma análoga, como visto na Fig. 2.2.1. A Fig 6(a) ilustra a definição matricial associada à uma transformação $H^{\otimes 2}$. Na Fig 6(b) tem-se a representação da transformação $H^{\otimes 2}$ aplicada a um estado quântico clássico $|00\rangle$. No *QuIDDPro*, e aplicação de uma transformação quântica é modelada por uma operação entre grafos.

$$
\begin{array}{c|c}
R_0R_1 & \\
00 & \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix} \\
01 & \\
10 & \\
11 & \\
 & \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\
 & C_0C_1
\end{array}
$$

(a) Definição matricial de $H^{\otimes 2}$.



(b) Representação de $H^{\otimes 2}|00\rangle$ usando um QuIDD.

Figura 6: Transformações quânticas no QuIDDPro (VIAMONTES, 2007)

Vários resultados obtidos pelo *QuIDDPro* são apresentados em (GROUP, 2007).

Como destaque, tem-se a simulação de instâncias do Algoritmo de Grover para sistemas de até 40 *qubits*, no qual o consumo de memória não ultrapassou $0,398$ MB, enquanto que demais pacotes de simulação ficaram limitados a sistemas de até 25 *qubits*. Entretanto, por se tratar de um processamento sequencial, a simulação demorou $8,23\ e^4$ segundos.

2.2.2 Massive Parallel Quantum Computer Simulator

O *Massive Parallel Quantum Computer Simulator* (*MPQCS*) (RAEDT et al., 2006) consiste em um *software* para simulação paralela de computadores quânticos, podendo ser aplicado a máquinas paralelas *high end* ou a *clusters* de *desktops* comuns. Implementado em Fortran 90, o simulador suporta as transformações quânticas universais necessárias para descrição de qualquer algoritmo quântico. O algoritmo pode ser descrito a partir de um pseudo-código, demonstrado na Figura 7, ou por um circuito quântico, gerado a partir de uma *interface* gráfica desenvolvida para *MS Windows*. O circuito é interpretado e automaticamente gera o correspondente pseudo-código.

```
QUBITS 32                    ! The command QUBITS sets the size of the quantum computer
INITIAL STATE 0              ! The initial state of the quantum computer is set to |0...0>
MPIPROCESSES 32              ! Number of MPI processes is set to 32
                             ! (not used by the OpenMP code)
H 0                          ! Hadamard operation on qubit 0
.                            ! These lines are omitted here but contain commands for
.                            ! Hadamard operations carried out on qubits 1-25
.
H 26                         ! Hadamard operation on qubit 26
SWAP 1 0 27                  ! Command to swap 1 pair of qubits: qubits 0 and 27
H 27                         ! Hadamard operation on qubit 27
SWAP 1 27 28                 ! Command to swap qubits 27 and 28
```

Figura 7: Pseudocódigo para descrição de uma aplicação quântica no Massive Parallel Quantum Computer Simulator (RAEDT et al., 2006)

A técnica de paralelização do *MPQCS* consiste em distribuir o espaço de estados do sistema quântico através de todos os nodos do *cluster*, utilizando o modelo de programação *MPI* para comunicação entre os processos. A quantidade de processos (N) *MPI* necessários para construção do sistema quântico depende da razão $N = 2^L/2^M$, onde L representa a quantidade de *qubits* do sistema e M é a quantidade de *qubits* que cada processo é capaz de armazenar. Dessa forma, cada processo têm acesso direto à um determinado intervalo do espaço de estados do sistema quântico, enquanto que os demais estados são acessíveis através de comunicação com os correspondentes processos. A Figura 8 ilustra a tabela que descreve um cenário com $L = 4$ e $M = 2$, associando os processos às respectivas amplitudes armazenadas.

Os valores **00**, **01**, **10** e **11** fazem alusão ao *rank* de cada processo MPI. Já *00*, *01*, *10* e *11* indexam as posições de memória de cada processo.

Considerando que a aplicação de uma transformação quântica sobre o sistema usualmente altera as amplitudes de todos os estados, percebe-se a necessidade de um método para obter as amplitudes armazenadas em processos diferentes. Exemplificando,

	$\sigma_1 = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}$			
	00	**01**	**10**	**11**
00	a(0000)	a(0100)	a(1000)	a(1100)
01	a(0001)	a(0101)	a(1001)	a(1101)
10	a(0010)	a(0110)	a(1010)	a(1110)
11	a(0011)	a(0111)	a(1011)	a(1111)

Figura 8: Tabela com a distribuição de amplitudes para 4 processos MPI (RAEDT et al., 2006)

considera-se a distribuição da tabela ilustrada pela Figura 8 e a aplicação da transformação quântica *NOT* ao *qubit* 2 do sistema. Para efetivação dessa operação, é necessário que cada processo *MPI* tenha conhecimento das amplitudes associadas aos estados $|0\rangle$ e $|1\rangle$ do *qubit* 2. Essa condição é satisfeita após a troca de amplitudes entre os processos, definida pela permutação σ_2, apresentada na Figura 9. Percebe-se, agora, que todos os processos possuem as amplitudes associadas aos estados do *qubit* 2. Assim, todos os processos podem aplicar, simultaneamente, a transformação quântica sobre esse *qubit*, caracterizando a evolução do estado global do sistema quântico. Descrições detalhadas de como essa troca é realizada podem ser obtidas em (RAEDT et al., 2006).

	$\sigma_1 = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}$				$\sigma_2 = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 0 & 1 & 2 \end{pmatrix}$			
	00	**01**	**10**	**11**	**00**	**01**	**10**	**11**
00	a(0000)	a(0100)	a(1000)	a(1100)	a(0000)	a(0001)	a(1000)	a(1001)
01	a(0001)	a(0101)	a(1001)	a(1101)	a(0100)	a(0101)	a(1100)	a(1101)
10	a(0010)	a(0110)	a(1010)	a(1110)	a(0010)	a(0011)	a(1010)	a(1011)
11	a(0011)	a(0111)	a(1011)	a(1111)	a(0110)	a(0111)	a(1110)	a(1111)

Figura 9: Tabela com a distribuição de amplitudes para N processos MPI considerando a permutação σ_2 (RAEDT et al., 2006)

Os testes foram realizados em vários supercomputadores, como IBM BlueGene/L, Cray X1E, IBM Regatta p690+, dentre outros. Os principais resultados demonstram a capacidade de simulação de sistemas com até 36 *qubits*, exigindo aproximadamente 1 TB de memória RAM e 4096 processadores. Esse alto custo se deve ao armazenamento explícito de todas as amplitudes que definem o estado do sistema quântico, implicando no armazenamento de 2^{36} valores complexos, usualmente representados por dois dados do tipo *float*.

Em 2010, o *MPQCS* fez uso do supercomputador *JUGENE* para executar uma instância do Algoritmo de Shor com 42 *qubits*, fatorando o número 15707 em 113×139 (HENKEL, 2010). Para isso, foram necessários 262.144 processadores, porém o consumo de memória e o tempo de simulação requerido não foram divulgados.

Dada a constante necessidade de comunicação entre os processos, o simulador exige

uma rede de intercomunicação de alta capacidade. Dessa forma, *clusters* comuns, formados por PCs conectados por uma rede *ethernet*, apresentarão uma limitação na quantidade de nodos que podem ser utilizados eficientemente, interferindo diretamente na dimensão dos sistemas quânticos suportados.

2.2.3 General-Purpose Parallel Simulator for Quantum Computing

A natureza inerentemente paralela associada à evolução de um sistema quântico é explorada, no *General-Purpose Parallel Simulator for Quantum Computing* (NIWA; MATSUMOTO; IMAI, 2008), a partir do particionamento da matriz associada à transformação unitária em sub matrizes. Essas submatrizes são aplicadas, de forma paralela, a sub vetores associados ao estado do sistema quântico.

Para a aplicação de uma transformação sobre o *qubit i*, como ilustrado pela Figura 10, faz-se necessário a geração da matriz X a partir do produto tensor $X = (\otimes_{k=1}^{i-1} I) \otimes U \otimes (\otimes_{k=i+1}^{n} I)$.

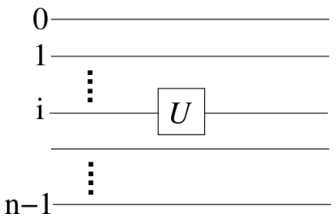


Figura 10: Transformação quântica de 1 *qubit*

A Figura 11 descreve uma representação genérica para a matriz X, na qual tem-se uma possível decomposição em termos de submatrizes S_k.

$$X = \underbrace{\begin{pmatrix} S_0 & & & & & 0 \\ & S_1 & & & & \\ & & \cdots & & & \\ & & & \cdots & & \\ & & & & S_{2^i-2} & \\ 0 & & & & & S_{2^i-1} \end{pmatrix}}_{2^n} \text{ where } S_k = \underbrace{\begin{pmatrix} u_{11} & & \mathbf{0} & u_{12} & & \mathbf{0} \\ & \cdots & & & \cdots & \\ \mathbf{0} & & u_{11} & \mathbf{0} & & u_{12} \\ u_{21} & & \mathbf{0} & u_{22} & & \mathbf{0} \\ & \cdots & & & \cdots & \\ \mathbf{0} & & u_{21} & \mathbf{0} & & u_{22} \end{pmatrix}}_{2^{n-i}} (0 \leq k < 2^i)$$

Figura 11: Matriz que define a evolução do sistema quântico (NIWA; MATSUMOTO; IMAI, 2008)

A metodologia para particionamento de X depende da quantidade de processadores disponíveis (2^P) no *cluster*. A matriz X é decomposta em um sequência de multiplicações de submatrizes-subvetores indexadas por $M_k(0 \leq k < 2^i)$. M_k é definido como sendo $S_k\phi_k$, ou seja, o produto $S_k(2^{n-i} \times 2^{n-i}) \times \phi_k(2^{n-i})$, onde ϕ_k é um vetor que armazena as amplitudes dos estados do sistema quântico. Como todos os produtos $M_k(0 \leq k < 2^i)$ são independentes, tem-se a possibilidade de execução simultânea de 2^{i-P} multiplicações de submatrizes-subvetores em cada processador, como exemplificado pela Figura 12. Ao final da execução de cada produto, tem-se uma primitiva de

sincronização para atualizar todas as amplitudes do vetor de estado do sistema quântico.

$$X|\phi\rangle = \begin{pmatrix} S_0 & & & 0 \\ & S_1 & & \\ & & \cdots & \\ 0 & & & S_{2^i-1} \end{pmatrix} \begin{pmatrix} \phi_0 \\ \phi_1 \\ \cdots \\ \phi_{2^i-2} \\ \phi_{2^i-1} \end{pmatrix} \begin{matrix} \} (processor\ 0) \\ \cdots \\ \cdots \\ \cdots \\ \} (processor\ 2^P) \end{matrix} \quad \text{where } \phi_k = \begin{pmatrix} \alpha_{k2^{n-i}} \\ \alpha_{k2^{n-i}+1} \\ \cdots \\ \alpha_{(k+1)2^{n-i}-2} \\ \alpha_{(k+1)2^{n-i}-1} \end{pmatrix} \quad (0 \le k < 2^i)$$

Figura 12: Decomposição da transformação quântica (NIWA; MATSUMOTO; IMAI, 2008)

Quando o número de submatrizes S_k é menor que a quantidade de processadores disponíveis ($2^i < 2^P$), ocorre um desbalanceamento no *cluster*, fazendo com que determinados processadores trabalhem de forma excessiva enquanto outros permanecem ociosos. Assim, as matrizes S_k devem ser decompostas para viabilizar a execução paralela. Essa decomposição se dá pela divisão de S_k em 2^{P+1} *chunks* de linhas, indexados por $R_j (0 \le j < 2^{P+1})$, as quais armazenam $2^{n-i-(P+1)}$ linhas de S_k.

As multiplicações de *chunks* indexados por R_j e R_{2^P+j} são mapeadas para o mesmo processador, conforme apresentado na Figura 13.

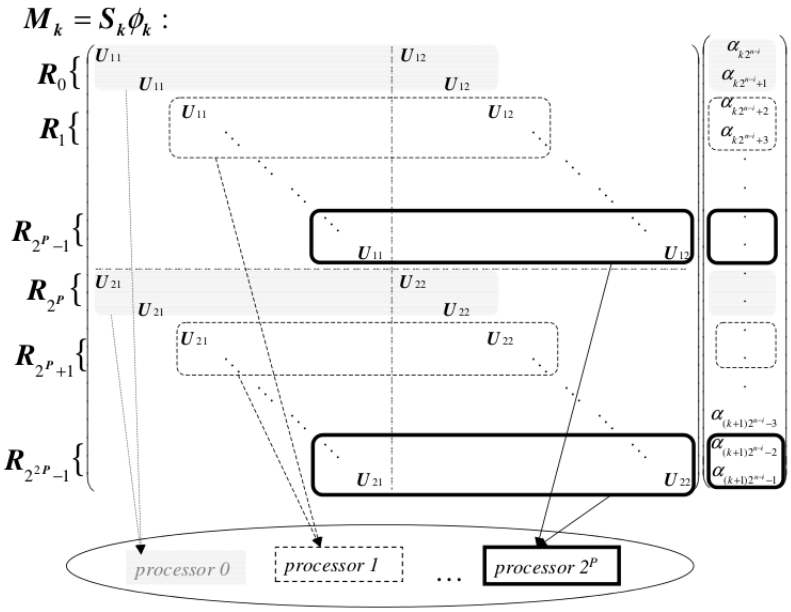


Figura 13: Decomposição de M_k e ϕ_k para casos em que $2^i \le 2^P$ (NIWA; MATSUMOTO; IMAI, 2008)

Transformações quânticas também podem ser definidas a partir de matrizes de rota-

ção ($U_R(\theta)$) e de mudança de fase ($U_{P1}(\phi)$ e $U_{P2}(\phi)$), definidas por

$$U_R(\theta) = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}, \quad U_{P1}(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} \quad e \quad U_{P2}(\phi) = \begin{pmatrix} e^{i\phi} & 0 \\ 0 & 1 \end{pmatrix}. \tag{26}$$

Por exemplo, a porta quântica *NOT* pode ser obtida por $U_R(\frac{\pi}{2})U_{P1}(\pi)$, permitindo a implementação de um modelo de erros, considerando a decoerência dos sistemas quânticos. Esse fenômeno pode ser simulado a partir da inserção de pequenos desvios nos ângulos θ e ϕ, permitindo uma simulação coerente com as teorias físicas que fundamentam a Teoria da Informação Quântica.

O simulador foi desenvolvido sobre o computador paralelo Sun Enterprise (E4500), o qual possui 8 processadores UltraSPARC-II (400MHz), 1 MB cache, 10GB RAM e OS Solaris 2.8 (64 *bits*).

Dentre os resultados obtidos, destaca-se a simulação de transformações *Hadamard* em um sistema quântico de 29 *qubits*, conforme apresentado na Figura 14.

Qubits	Num. of Procs			
	1	2	4	8
20	2.38	1.18	0.76	0.40
22	10.85	5.73	3.20	1.35
24	46.94	24.96	13.40	9.58
26	205.81	109.97	58.83	38.71
28	887.40	467.71	253.82	167.31
29	2027.9	1081.1	592.08	395.81

Figura 14: Tempos, em segundos, para simulação de transformações *Hadamard* (NIWA; MATSUMOTO; IMAI, 2008)

O simulador fica limitado em aproximadamente 29 *qubits* devido ao custo de armazenamento do espaço de estados e da matriz de transformação do sistema, os quais crescem exponencialmente conforme são adicionados novos *qubits*.

2.2.4 Quantum Computer Simulation Using the CUDA Programming Language

O simulador quântico descrito em (GUTIERREZ et al., 2010) utiliza as premissas do modelo de programação CUDA para exploração do paralelismo associado a evolução dos sistemas quânticos. Dessa forma, viabiliza-se a execução do cálculo das amplitudes associadas aos estados da base computacional a partir de um grande número de *threads*, as quais executam sobre as unidades de processamento das GPUs.

Essa abordagem considera o armazenamento explícito de todas as amplitudes que definem o estado do sistema quântico, de forma que uma grande quantidade de memória é necessária para suporte a simulação de algoritmos complexos. A obtenção das novas amplitudes dos estados não se dá por matrizes geradas a partir da operação de produto

tensor, reduzindo o custo de processamento e armazenamento da simulação. As novas amplitudes são definidas a partir das matrizes associadas às transformações quânticas universais, de um e dois *qubits*. Essas matrizes fornecem os coeficientes que serão multiplicados pelas amplitudes associadas aos estados do sistema. A descrição detalhada do cálculo das amplitudes pode ser estudado em (GUTIERREZ et al., 2010).

Para realização dos cálculos, é necessário copiar todo o vetor de amplitudes para a memória global da GPU. Visando um acesso mais rápido a esses dados durante a execução das *threads*, os conjuntos de amplitudes a serem modificadas são copiadas para a área de memória compartilhada, a qual possui menor tempo de acesso. Entretanto, faz-se necessário uma função de mapeamento que realize essa cópia de forma eficiente, garantindo um acesso coalescido à memória, aproveitando a largura de banda disponível no barramento e evitando a serialização das *threads*.

Cada bloco de *threads* é associado com determinados conjuntos de amplitudes (*closed groups*), os quais podem ser processados de forma independente. Cada *thread* do bloco realiza, simultaneamente, a computação associada a determinadas amplitudes contida nos conjuntos. Esse mapeamento depende da granulosidade do problema e da quantidade de recursos computacionais disponíveis. Dessa forma, pode ser definido previamente que uma *thread* compute apenas algumas amplitudes ou todas as amplitudes de vários conjuntos distintos.

A análise de desempenho considera uma GPU NVIDIA GeForce 8800GTX e CUDA 1.1. As principais características da GPU são descritas na Tabela 2, destacando a quantidade total de processadores disponíveis (128), de forma a viabilizar a execução paralela de muitas *threads*. O espaço de memória disponível (768 MB) é a principal limitação para realização das computações, uma vez que o espaço de estados dos sistemas quânticos é exponencialmente maior que a quantidade de *qubits* utilizados para modelar o sistema.

Tabela 2: Principais características da GeForce 8800GTX

Atributo	Valor
Multiprocessadores	16
Processadores em cada multiprocessador	8
Frequência de *clock*	1,35 GHz
Memória global	768 MB

Cada amplitude é um valor complexo representado por 2 *floats* de 32 *bits* cada, permitindo o armazenamento de até 26 *qubits* (2^{26} amplitudes) no espaço de memória global da GPU. Devido ao tamanho reduzido da memória compartilhada, é possível realizar a cópia de até 2^{10} amplitudes, limitando a quantidade de amplitudes a serem calculadas simultaneamente. Uma análise geral de desempenho é vista na Figura 15.

As simulações sequenciais realizadas consideram um PC com Intel Core2Duo 6400

n	CUDA programming model on GPU						Sequential code on CPU		Speed-up t_{CPU}/t_{GPU}
	Gate-by-gate t (msec)	Coalescing-aware					Libquantum t (msec)	Optimized t (msec)	
		r	c	nstg	last	t (msec)			
Walsh									
15	0.32	10	4	2	5/6	0.11	4.42	1.76	16.3
16	0.46	10	4	2	6/6	0.20	10.57	3.82	18.9
17	0.78	9	4	3	3/5	0.42	32.42	7.90	18.7
18	1.39	9	4	3	4/5	0.81	83.07	18.82	23.1
19	2.68	9	4	3	5/5	1.63	179.72	47.89	29.4
20	5.41	9	5	4	3/4	3.48	380.81	100.45	28.9
21	11.07	9	5	4	4/4	7.15	825.84	210.78	29.5
22	22.82	9	5	5	1/4	15.41	1688.36	439.86	28.5
23	47.28	9	5	5	2/4	31.73	3364.75	919.95	29.0
24	98.26	9	5	5	3/4	65.39	7066.15	1927.47	29.5
25	203.69	9	5	5	4/4	135.15	15 077.27	4052.66	30.0
26	422.36	9	5	6	1/4	290.08	32 729.87	8774.42	30.2
QFT									
15	0.46	10	4	2	5/6	0.13	5.35	10.58	82.7
16	0.63	10	4	2	6/6	0.24	13.07	18.86	77.0
17	0.98	9	5	3	4/4	0.52	42.67	40.09	77.8
18	1.62	9	4	3	4/5	0.98	120.39	86.90	88.6
19	2.99	9	4	3	5/5	1.96	254.19	187.59	95.8
20	5.78	8	4	4	4/4	4.14	520.33	389.25	93.9
21	11.60	9	5	4	4/4	8.57	1107.78	816.98	95.3
22	23.71	8	4	5	2/4	18.26	2264.26	1715.42	93.9
23	48.84	8	4	5	3/4	37.67	4631.77	3588.86	95.3
24	101.04	9	4	4	5/5	76.60	9752.44	7492.01	97.8
25	209.12	9	5	5	4/4	161.58	22 878.99	15 642.21	96.8
26	433.03	8	4	6	2/4	343.30	55 325.69	32 692.91	95.2

Figura 15: Tempos de simulação para transformações Walsh-Hadamard e QFT (GUTIERREZ et al., 2010)

@ 2,13GHz com 2GB RAM, utilizando a biblioteca *libquantum* para simulação quântica. Como principais conclusões, destaca-se que o tamanho da memória física da GPU limita de forma significativa a quantidade de *qubits* suportados. Entretanto, para sistemas com até 26 *qubits*, um *speedup* de até $95\times$ pode ser obtido, caracterizando um excelente ganho de desempenho.

2.3 Considerações Finais

O estudo dos fundamentos da *CQ* é essencial para compreensão dos fenômenos que proporcionam a alta capacidade de processamento previsto por esse paradigma computacional. A breve descrição dos postulados da *MQ*, realizada neste Capítulo, introduz os conceitos fundamentais exigidos para interpretação dos algoritmos quânticos e, também, da correspondente modelagem e simulação.

A introdução de diferentes soluções para lidar com a complexidade de simulação de algoritmos quânticos a partir de computadores clássicos têm contribuído para delinear as melhores abordagens para lidar com os diferentes problemas decorrentes do aumento exponencial no armazenamento e processamento requeridos por sistemas com muitos *qubits*.

A proposta considerada pelo *QuIDDPro* é significante pela capacidade de redução no consumo de memória, viabilizando a simulação de sistemas com até 40 *qubits*. O *QuIDDPro* apresenta bons resultados quando existem padrões repetidos na estrutura matricial, ou seja, é eficiente em determinadas classes de algoritmos e pode não apresentar a mesma eficiência para casos genéricos. Apesar do ganho de memória obtido,

em determinadas configurações de transformações quânticas o tempo de simulação ainda permanece alto. Esse obstáculo pode ser superado através da paralelização das tarefas.

Os simuladores paralelos descritos neste capítulo representam transformações e estados quânticos explicitamente a partir de matrizes e vetores, respectivamente. Dessa forma, as possibilidades de simulação são limitadas pela quantidade de memória disponível do *cluster* utilizado. Entretanto, é possível obter ganho no tempo de simulação pela paralelização do produto matriz-vetor que caracteriza a evolução de estado do sistema quântico. A distribuição das amplitudes dos estados ao longo dos nodos também é uma estratégia adotada, porém apresenta elevado custo de comunicação.

Uma abordagem ainda não consolidada é a integração desses dois esforços: (i) a busca por otimizações das estruturas associadas a transformações e estados quânticos; (ii) a paralelização das operações de evolução do estado do sistema considerando tais otimizações. A união dessas abordagens é um grande desafio, porém a perspectiva de bons resultados é uma grande motivação na busca por soluções mais eficientes para simulação de algoritmos quânticos a partir de computadores clássicos.

3 GPGPU: COMPUTAÇÕES GENÉRICAS EM GPU

Este capítulo apresenta detalhes arquiteturais das *GPUs* e descreve os principais conceitos dos modelos de programação CUDA e PyCUDA, utilizados neste trabalho para estabelecer o suporte à simulação quântica paralela.

3.1 Visão Geral

O paradigma de programação *GPGPU* (*General Purpose Computing on Graphics Processing Units*) está se consolidando como uma das abordagens mais difundidas para *HPC* (*High Performance Computing*) devido ao seu bom custo-benefício. Essa forma de computação explora o poder computacional atualmente encontrado nas GPUs (*Graphic Processing Units*) para realizar computações científicas de propósito geral. Tal poder computacional se dá, em primeira instância, pelo elevado número de *cores* encontrados nas *GPUs*, superando as arquiteturas *multi-core* nas quais as *CPUs* atualmente são baseadas. Consequentemente, a arquitetura das *GPUs* apresenta uma capacidade de processamento por dispositivo maior do que as melhores *CPUs* atuais.

Vale destacar que a diferença de desempenho entre *GPUs* e *CPUs* é mais acentuada quando realizadas computações sobre números de precisão simples. Para valores de precisão dupla, o desempenho da *GPU* é bastante reduzido visto que a quantidade de núcleos de processamento voltados à operações de precisão simples é maior do que os voltados à operações de precisão dupla.

Quando utilizada a abordagem *GPGPU*, geralmente utiliza-se a *CPU* como uma unidade de controle de fluxo das computações intensivas a serem executadas pela *GPU*. Para determinados problemas, a exploração do alto grau de paralelismo e poder computacional providos pelas *GPUs* viabiliza a aceleração de vários algoritmos, incluindo os relacionados à Computação Quântica.

As principais características arquiteturais que resultam na alta capacidade de processamento das *GPUs* são detalhadas a seguir.

3.2 Arquitetura

A arquitetura paralela das *GPUs* oferece alto poder computacional e largura de banda de memória, sendo adequada para acelerar vários tipos de aplicações (NVIDIA, 2012c). O alto desempenho está relacionado ao alto número de processadores (cores) residentes na GPU. A arquitetura é composta de *SMs* (*Streaming Multiprocessors*), controladores de memória, registradores, *CUDA cores* e diferentes espaços de memória.

Na arquitetura *Kepler* (NVIDIA, 2012a), a qual é a mais recente disponibilizada pela NVIDIA, são distribuídos de acordo com o diagrama de blocos ilustrado na Figura 16.

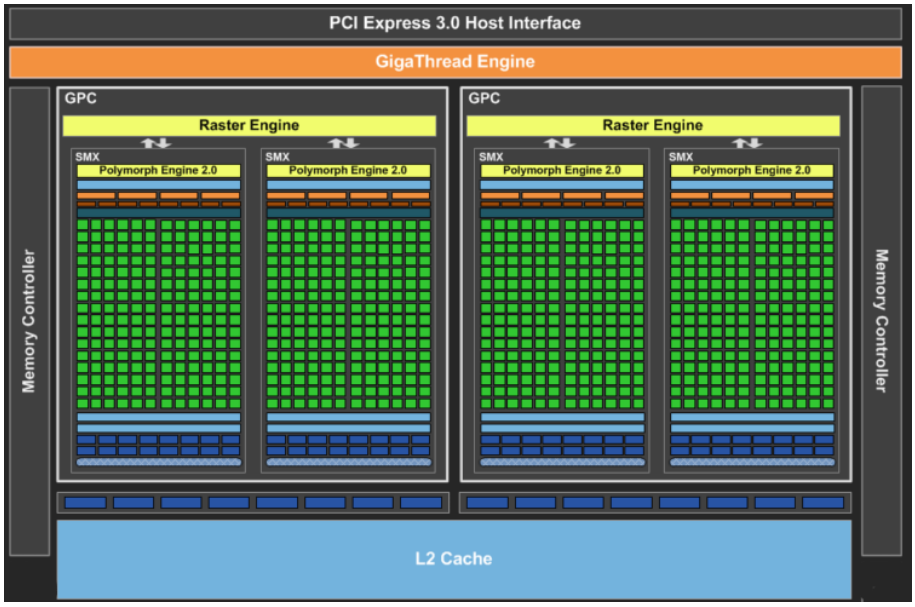


Figura 16: Diagrama de blocos da arquitetura Kepler (NVIDIA, 2012a)

Dentre os componentes apresentados na Figura 16, os seguintes se destacam:

- Memória Global: É o maior espaço para leitura e armazenamento de dados na placa de vídeo, disponibilizando uma interface de comunicação com um elevado *throughput* (até 140GB/s para placas com $1GB$ RAM) que permite a cópia de dados da memória global para a memória compartilhada. Entretanto, devido à alta latência ($\approx 400 - 800$ ciclos de *clock*) para acesso a um dado na memória global, a leitura e/ou cópia de dados deve ser planejada para que múltiplas *threads* acessem simultaneamente a memória;

- GPC (*Graphic Processing Cluster*): Consiste no maior bloco arquitetural que agrupa todos os componentes necessários para realizar uma computação na *GPU*. Um GPC possui seu próprio controlador de memória que atende à requisições de acesso a dados na memória global da *GPU* gerados pelas *threads* em execução no seu conjunto de *cores*;

- SM (*Streaming Multiprocessors*): É o agrupamento de componentes internos a cada GPC. O SM contém os *cores* para execução das computações gráficas e científicas e unidades para leitura e escrita na memória. A Tabela 3 descreve os componentes presentes em um SM.

Tabela 3: Componentes presentes em cada SM

Componente	Quantidade
CUDA Cores	192
LD/ST	32
Escalonadores	4
Unidades de Despacho	8
SFU	32
Registradores	65,536
Memória Compartilhada/Cache L1	64KB
Cache Memória Constante	8KB
Cache Memória de Textura	6-8KB

Um dos recursos mais críticos a serem gerenciados quando da execução de computações sobre *GPUs* é o correto uso dos diferentes espaços de memória disponíveis. A memória compartilhada (*Shared Memory*) atua como uma memória *cache* **gerenciada pelo programador**, de forma que determinados padrões de comportamento da aplicação em execução possam ser explorados para determinar quais valores serão acessados. Diferentemente da memória global, que possui um elevado tempo de acesso, a memória compartilhada pode ser tão rápida quanto os registradores, desde que determinados padrões de acesso sejam garantidos para que conflitos de acesso sejam evitados (veja detalhes na Seção 3.5).

A forma mais simples para utilização da memória compartilhada segue (mas não fica limitada a essas funcionalidades) os seguintes passos básicos:

- Copiar os dados da memória global para a memória compartilhada;
- Sincronizar as *threads* para garantir a consistência dos dados;
- Realizar as computações sobre os dados da memória compartilhada;
- Se necessário, sincronizar novamente as *threads* para que todos os resultados sejam escritos corretamente;
- Copiar os resultados para a memória global.

É importante destacar a importância de projetar detalhadamente cada uma das etapas mencionadas, uma vez que tanto as computações quanto as cópias de dados devem

seguir determinadas regras para que a capacidade de processamento da *GPU* seja aproveitada ao máximo. Detalhes das otimizações a serem consideradas poder ser encontradas em (NVIDIA, 2012b).

Conforme descrito na Tabela 3, cada SM possui ainda 3 áreas de memória cache **gerenciadas pela aplicação**: cache para memória constante (*Constant Cache*), cache para memória de textura (*Texture Cache*) e cache L1 (*L1 Cache*).

Constant Cache: Cache de leitura compartilhada por todas unidades funcionais de um mesmo SM, utilizada para acelerar o acesso a dados da memória constante. A memória constante, com tamanho máximo de $64KB$, localiza-se em um espaço reservado da **memória global** da *GPU*. Consequentemente, quando da ocorrência de um *cache miss*, a latência de acesso ao dado na memória constante é equivalente ao acesso à memória global.

Texture Cache: Cache de leitura para a memória de textura. A vantagem associada a memória de textura é que sua cache possui *hardware* dedicado para aplicação de determinados filtros aos valores no momento de sua leitura. Para problemas nos quais a localidade espacial no sistema de coordenadas considerado para indexação dos elementos é essencial, e os filtros disponíveis são aplicáveis, o uso dessa memória e suas opções de indexação são capazes de resultar em uma execução mais eficiente.

L1 Cache: A cache L1 é individual a cada SM e é utilizada para suprir requisições de acesso à variáveis locais [1] que não estão armazenadas em registradores.

3.3 CUDA Framework

O modelo de programação paralelo *CUDA* estende algumas funcionalidades da linguagem *C* de forma a disponibilizar um *framework* de desenvolvimento de mais alto nível para programação paralela em *GPUs*. Seguindo uma abordagem SIMT (*Single Instruction Muliple Threads*), esse modelo prevê que no mesmo ciclo de *clock* várias *threads*, agrupadas em conjuntos de 32, denominadas *warps*, estarão executando a mesma instrução.

O modelo provê abstrações como *threads*, *grids*, memória compartilhada e barreiras de sincronização que ajudam o programador a explorar de forma mais eficiente todos os recursos disponíveis na *GPU*. O código consiste em um *host-code* e um *device-code*.

O *host-code* é executado pela *CPU* e contém a porção sequencial e com pouca computação do algoritmo a ser implementado. É geralmente utilizado para preparar as estruturas que serão utilizadas durante a execução na *GPU* e eventualmente pode

[1] Neste contexto, "locais" indica que a variável é privada a uma dada *thread*, porém seu armazenamento se dá na memória global.

apresentar alguma etapa de pré-processamento. Em sua versão mais simples, esse código consiste nas seguintes etapas:

- Criação das estruturas de dados a serem computadas;
- Cópia dessas estruturas para a memória da *GPU*;
- Invocação da função (*kernel*) a ser executada pela *GPU*;
- Cópia dos resultados gerados pela *GPU*.

O *device-code*, ou *kernel*, é executado diretamente na GPU e caracterizado por ser a porção paralela e computacionalmente intensiva do problema a ser solucionado. O *kernel* é uma função em *C* que, quando invocada pelo *host-code*, é executado N vezes em paralelo por N *CUDA threads* diferentes. O especificador **__global__** identifica que um determinado método dentro do código é um *kernel* CUDA.

No *host-code*, quando da invocação de um *kernel*, são especificadas as *configurações de execução* a partir da sintaxe $<<<>>>$, definindo como as *threads* serão organizadas em termos de blocos de *threads* e *grids*.

Um exemplo de código para soma de vetores no formato $C = A + B$ é apresentado no Código 1. Destaca-se que apenas os elementos essenciais para execução foram considerados. Verificações de erro associados à alocação de memória e à cópia de dados foram intencionalmente omitidos.

Código 1: Exemplo de código para soma de vetores utilizando CUDA

```
1  #include <stdio.h>
2  #include <cuda_runtime.h>
3
4  /** Código do CUDA Kernel
5   * Realiza a computação C = A + B. Os 3 vetores possuem o mesmo tamanho '
        numElements'.
6   */
7  __global__ void
8  vectorAdd(const float *A, const float *B, float *C, int numElements)
9  {
10     int i = blockDim.x * blockIdx.x + threadIdx.x;
11     if (i < numElements)
12         C[i] = A[i] + B[i];
13 }
14
15 // Código executado na CPU
16 int
17 main(void)
18 {
19     int numElements = 50000;
20     size_t size = numElements * sizeof(float);
21
22     float *h_A = (float *)malloc(size); // Aloca o vetor A na memória do host
23     float *h_B = (float *)malloc(size); // Aloca o vetor B na memória do host
24     float *h_C = (float *)malloc(size); // Aloca o vetor C na memória do host
25
```

```
26      // Inicializa os vetores A e B com valores aleatórios
27      for (int i = 0; i < numElements; ++i)
28      {
29          h_A[i] = rand()/(float)RAND_MAX;
30          h_B[i] = rand()/(float)RAND_MAX;
31      }
32
33      // Aloca o espaço de memória na GPU para o vetor A
34      float *d_A = NULL;
35      cudaMalloc((void **)&d_A, size);
36
37      // Aloca o espaço de memória na GPU para o vetor B
38      float *d_B = NULL;
39      cudaMalloc((void **)&d_B, size);
40
41      // Aloca o espaço de memória na GPU para o vetor C
42      float *d_C = NULL;
43      cudaMalloc((void **)&d_C, size);
44
45      cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice); // Copia o vetor A para a
            GPU
46      cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice); // Copia o vetor B para a
            GPU
47
48      // Definição dos parâmetros de configuração da execução na GPU
49      int threadsPerBlock = 256;
50      int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
51      vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
52
53      cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost); // Copia o resultado
            armazenado na GPU para o vetor C na CPU
54
55          // Desaloca os vetores na GPU
56      cudaFree(d_A);
57      cudaFree(d_B);
58      cudaFree(d_C);
59
60      // Desaloca os vetores na CPU
61      free(h_A);
62      free(h_B);
63      free(h_C);
64
65      cudaDeviceReset(); // Reseta a GPU e finaliza
66      return 0;
67  }
```

3.3.1 Hierarquia de Threads

A organização das *threads* considera uma hierarquia de blocos e *grids* que são configurados de acordo com o problema a ser resolvido, conforme ilustrado na Figura 17. Um bloco agrupa várias *threads* sob um mesmo conjunto de índices **blockIdx.x** e **blockIdx.y**. A quantidade de *threads* residentes em um mesmo bloco é definida de acordo com os dados a serem processados pelo algoritmo e considera as limitações do *hardware* utilizado. Para a atual geração de *GPUs*, um bloco fica limitado a 1024 *threads*, uma vez que todas as *threads* de um mesmo bloco devem ser alocadas a um mesmo SM.

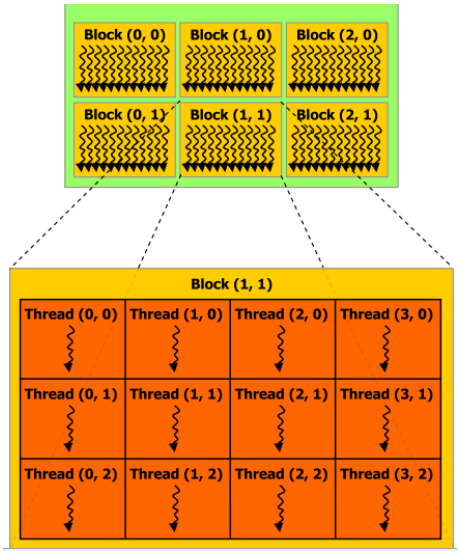


Figura 17: Agrupamento de *threads* em termos de *grids* e blocos (NVIDIA, 2012b)

Outro parâmetro que influencia a configuração de um bloco diz respeito ao uso da memória compartilhada, a qual pode ser utilizada pelas *threads* de um mesmo bloco para se comunicar e partilhar resultados. Como a memória compartilhada é um recurso com tamanho reduzido, um bloco com muitas *threads* pode requisitar todo esse espaço compartilhado, impedindo que outros blocos sejam executados simultaneamente.

Blocos de *threads* devem apresentar um comportamento tal que possam ser executados em qualquer ordem e de forma independente, gerando sempre o mesmo resultado. Essa independência relacionada ao fluxo de execução deve ser garantida uma vez que os blocos de *threads* são alocados aos SM de várias formas, dependendo do *hardware* disponível.

3.4 PyCUDA

Embora o modelo de programação *CUDA* seja baseado na linguagem *C/C++*, outras linguagens e bibliotecas são suportadas. No contexto deste trabalho, a extensão para a linguagem *Python*, chamada *PyCUDA* (KLOECKNER, 2012), foi escolhida devido a duas razões principais:

- A prototipação com a linguagem *Python* resulta em um desenvolvimento simplificado e mais ágil devido às poucas restrições da linguagem;

- O *host-code* é compreendido por métodos para a criação das estruturas básicas que são posteriormente copiadas para a *GPU*. Especificamente neste trabalho, tal criação é baseada em uma formatação de *strings* e na manipulação de estruturas multidimensionais, as quais são facilmente manipuladas com *Python*. Já o *device-code* executa uma computação mais restrita e intensiva, a qual é implementada

na linguagem *C* como um *CUDA kernel* tradicional.

Pela exploração de funcionalidades do *PyCUDA*, como *garbage collection*, mensagens de erro simplificadas e desenvolvimento mais rápido, a parte técnica do desenvolvimento torna-se mais fácil e uma maior atenção pode ser dada ao problema algorítmico. Um *workflow* de desenvolvimento básico utilizando *PyCUDA* é apresentado na Figura 18. Executáveis binários, obtidos a partir de um código fonte associado a um *CUDA kernel*, são gerados pelo compilador do *PyCUDA*. Como o *CUDA kernel* é basicamente uma *string* dentro do código *Python*, tem-se a possibilidade de formação dessa *string* durante a execução do algoritmo. Consequentemente, tem-se a possibilidade de geração de código em tempo de execução, onde o *kernel* é compilado e armazenado em uma cache semi-permanente para posterior reuso, caso o código fonte não tenha sido modificado.

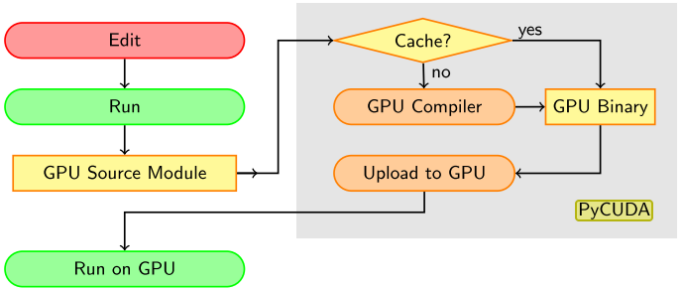


Figura 18: Workflow de compilação do PyCUDA (KLOECKNER, 2012)

Como exemplo de uso da biblioteca *PyCUDA*, no Código 2 é descrito um código que realiza a soma de vetores no formato $C = A + B$ de forma similar ao Código 1, porém agora utilizando os recursos providos pela biblioteca *PyCUDA*.

Código 2: Exemplo de código para soma de vetores utilizando PyCUDA

```
1  import pycuda.autoinit
2  import pycuda.driver as drv
3  import numpy
4  from pycuda.compiler import SourceModule
5
6  # Kernel a ser executado na GPU
7  mod = SourceModule("""
8  __global__ void vectorADD(float *C, float *A, float *B)
9  {
10   const int i = threadIdx.x;
11   C[i] = A[i] + B[i];
12 }
13 """)
14
15 vectorADD = mod.get_function("vectorADD")
16
17 A = numpy.random.randn(400).astype(numpy.float32) # Inicialização do vetor A na
       memória do host
18 B = numpy.random.randn(400).astype(numpy.float32) # Inicialização do vetor B na
       memória do host
19 C = numpy.zeros_like(a) # Inicialização do vetor C na memória do host, com todos
       valores zerados
```

```
20
21 # Invocação do kernel com 400 threads
22 multiply_them(
23         drv.Out(C), drv.In(A), drv.In(B),
24         block=(400,1,1), grid=(1,1))
25
26 print C # Impressão do resultado do vetor C
```

É possível perceber que a implementação utilizando *PyCUDA* se mostra mais simples, porém é igualmente eficiente. A alocação, inicialização e cópia dos vetores é feita a partir de métodos da linguagem *Python*. Já o *CUDA kernel*, denominado *vectorADD*, tem sua definição na linguagem *C* e é o mesmo utilizado no Código 1.

3.5 Políticas de Otimização

Apesar da aparente simplicidade expressada pelos exemplos de soma de vetores apresentados nas Seções 3.4 e 3.3, o pleno potencial de processamento das *GPUs* somente será atingido se uma série de condições forem satisfeitas. Essas condições são comumente relacionadas ao acesso à memória (global, constante, compartilhada, ...), dado que tal ação resulta em uma interrupção do fluxo de execução de computações de uma *thread* e, consequentemente, na possibilidade da ocorrência de um estado de ociosidade em *cores* do SM.

Conforme já introduzido, as *threads* em um bloco são agrupadas em *warps* e executam a mesma instrução em um ciclo de *clock*. Quando uma *warp* ($w1$) executa uma instrução de acesso a um dado na memória global da *GPU* (leitura ou escrita), tem-se uma interrupção do uso dos *cores*. Como a latência de acesso a memória global é de aproximadamente $400 - 800$ ciclos de *clock*, existe a possibilidade de iniciar a execução de outra *warp* ($w2$) sobre os *cores* ociosos até que a operação de acesso à memória de $w1$ seja concluída. Essa estratégia é denominada **latency hiding**, na qual busca-se minimizar os atrasos relacionados ao acesso a memória pela execução simultânea de várias *warps*. Dessa forma, é possível manter ocupados tanto os recursos de memória (unidades de *load*/*store*, controladores de memória, barramentos) quanto os recursos de processamento (*cores* e *SFUs*).

Quando acessos à memória global não podem ser evitados, a melhor estratégia para aproveitar a elevada largura de banda disponibilizada é garantir que as *threads* acessem os dados de forma **coalescida**. O acesso coalescido ocorre quando *threads* de uma mesma *warp* acessam endereços de memória consecutivos e múltiplos do tamanho do dado acessado (32, 64 ou 128 bytes), de forma a atender todas as 32 requisições com apenas uma transação de memória. Caso contrário, novas transações de memória serão geradas para suprir todas as requisições, gerando transferências de dados que não serão utilizados pelas *threads* e reduzindo o *throughput* de dados.

O acesso coalescido aos dados é garantido quando as *threads* apresentam determi-

nados **padrões acesso**. Nas Figuras 19 e 20 são ilustrados exemplos de acesso nos quais uma *warp* necessita de 128 *bytes* de dados, os quais estão armazenados em um intervalo contíguo e com início em um endereço múltiplo de 128. Independente se as *threads* acessam endereços consecutivos nesse intervalo ou não, apenas **1 transação de memória** que transfere 128 *bytes* atende a todas as requisições da *warp*.

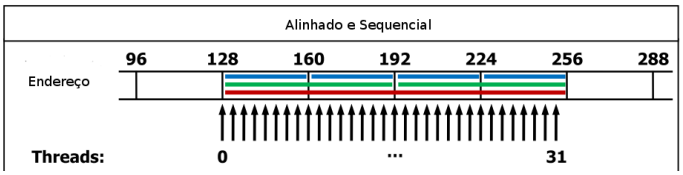


Figura 19: Acesso coalescido e sequencial

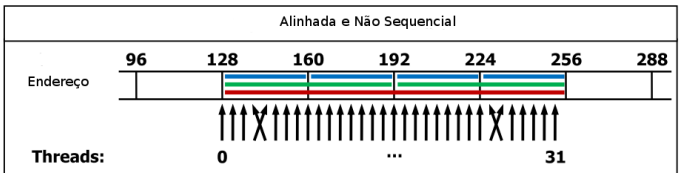


Figura 20: Acesso coalescido e não sequencial

Quando o padrão de acesso das *threads* não coincide com a forma como os dados estão armazenados na memória global, perdas de performance ocorrerão devido à necessidade de mais transações de memória. O exemplo apresentado na Figura 21 ilustra um cenário no qual primeiro endereço (132) indexado pela *warp* não é múltiplo de 128. Nesse caso, duas transações de memória serão realizadas: $(T1)$ para obter os dados no intervalo $[128, 256]$; e $(T2)$ para obter os dados no intervalo $[256, 384]$. Para $(T1)$, 124 dos 128 bytes transferidos são utilizados pelas *threads*. Entretanto, dentre os 128 bytes transferidos por $(T2)$, apenas 4 (referentes ao endereço 256) são de fato utilizados.

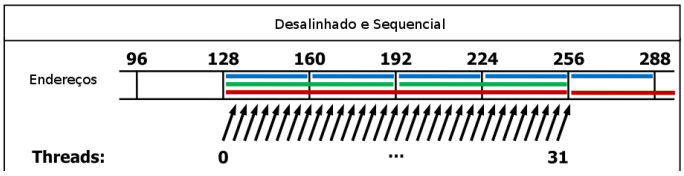


Figura 21: Padrão de acesso desalinhado, gerando duas transações de memória.

Quando considerados cenários nos quais o padrão de acesso é aleatório e não considera endereços de memória consecutivos, uma quantidade maior de transações se faz necessária, afetando de forma significativa o desempenho da aplicação.

Como já mencionado em seções anteriores, a memória compartilhada possui latência extremamente baixa e é ideal para ser utilizada como uma *cache* gerenciada pelo programador, armazenando temporariamente dados que são copiados de/para a memória

global da *GPU*. Entretanto, para que o desempenho dessa memória seja plenamente alcançado, algumas condições de acesso devem ser garantidas.

A memória compartilhada é dividida em 32 bancos que armazenam dados de 4 ou 8 bytes (configurável pelo programador), e dados consecutivos são armazenados em diferentes bancos. Esse comportamento é ilustrado na Figura 22, no qual, para exemplificação, são considerados 4 bancos ao invés de 32. Para o vetor de dados contendo elementos de 4 bytes, a distribuição se dá conforme a ilustração. Dados consecutivos são armazenados em bancos diferentes. Ao atingir o último banco disponível, o dado seguinte é armazenado na segunda coluna do primeiro banco, e a alocação segue esse padrão até que todos os valores sejam armazenados.

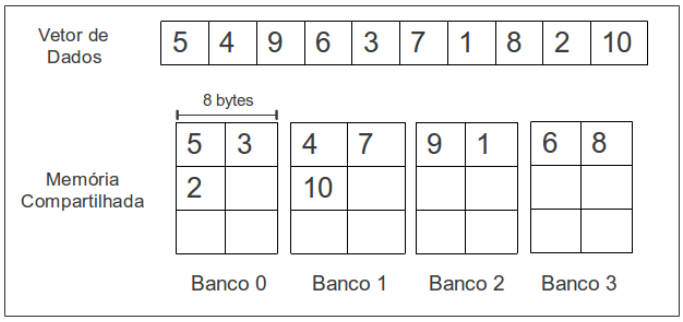
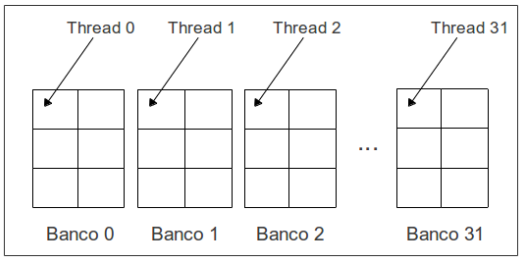


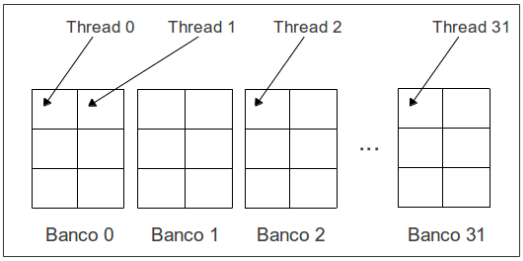
Figura 22: Forma de alocação de dados na memória compartilhada

O desempenho da memória compartilhada será explorado em sua plenitude, ou seja, requisições serão atendidas com apenas uma transação de memória, caso não ocorram conflitos de acesso aos bancos de memória (*bank conflicts*). Os seguintes cenários, ilustrados na Figura 23 são livres de conflito:

- Requisições de uma *warp* que resultam no acesso a 32 bancos diferentes (Figura 23(a));
- Requisições de uma *warp* que resultam no acesso a dois dados diferentes em uma mesma linha de um banco (Figura 23(b));



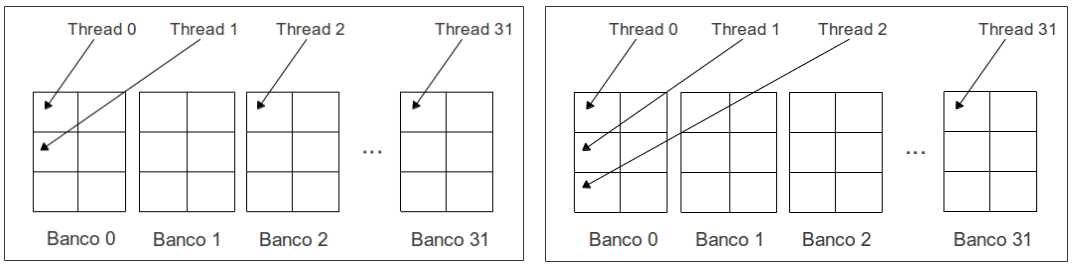
(a) Acesso a bancos diferentes.



(b) Acesso a dados diferentes em uma mesma linha de um banco.

Figura 23: Cenários nos quais não ocorrem conflito de acesso a um banco de memória

Quando requisições indexam dados armazenados em linhas diferentes de um mesmo banco, conflitos de acesso ocorrem. Consequentemente, tem-se um aumento na latência de acesso e redução do *throughput* uma vez que os acessos são serializados para atender todas as *threads*. Exemplos de conflitos são ilustrados na Figura 24.



(a) *One-way conflict*. (b) *Two-way conflict*.

Figura 24: Cenários nos quais ocorrem conflito de acesso a um banco de memória

3.6 Considerações Finais

A área de processamento de alto desempenho oferece uma série de soluções para o tratamento de problemas computacionais complexos. Entretanto, algumas dessas soluções exigem sistemas computacionais elaborados de alto custo de aquisição e manutenção, sendo uma opção inviável para grupos de pesquisa com recursos financeiros limitados.

Com o advento do paradigma *GPGPU* e o bom custo $\times$ benefício das placas de vídeo atuais, a utilização de *frameworks* de desenvolvimento (CUDA,OpenCL) voltados para processamento em *GPUs* apresenta-se como uma abordagem interessante que possibilita a aceleração de computações de vários algoritmos, incluindo dos relacionados à simulação de algoritmos da computação quântica.

A introdução das extensões de programação da plataforma CUDA oferecem abstrações para que programadores possam projetar algoritmos que exploram todos os recursos disponibilizados pelas *GPUs*. Visando a flexibilização no desenvolvimento das aplicações, extensões para a integração da plataforma CUDA com bibliotecas e linguagens de programação variadas estão em contínuo desenvolvimento.

A extensão *PyCUDA* é caracterizada pela possibilidade de desenvolvimento de programas na linguagem *Python*, oferecendo métodos para acesso aos recursos internos das *GPUs*. O *kernel* correspondente à porção intensiva da computação é descrito na linguagem *C* e compilado em tempo de execução, sendo esta tarefa gerenciada pelo interpretador do *PyCUDA*.

Apesar das facilidades proporcionadas pelas extensões já disponibilizadas para a plataforma CUDA, características arquiteturais das *GPUs* devem ser estudadas de forma cuidadosa para que os algoritmos desenvolvidos se adequem aos recursos disponibilizados.

Estratégias de otimização popularmente estudadas na área de *GPGPU* tem como foco principal garantir um acesso eficiente a memória, uma vez que este é um dos principais limitadores de desempenho quando da execução de computações sobre *GPUs*.

Com metodologias de desenvolvimento bem definidas e uma clara visão arquitetural das *GPUs*, é possível obter algoritmos paralelos com desempenho muito superior à versões tradicionalmente voltadas para o processamento com *CPUs*, exigindo baixo investimento financeiro para aquisição de *hardware* e, em alguns casos, apresentando eficiência energética superior à versões executadas a partir de *CPUs* tradicionais.

4 PROCESSOS QUÂNTICOS: UMA NOVA INTERPRETAÇÃO PARA TRANSFORMAÇÕES QUÂNTICAS

Neste Capítulo são apresentadas as contribuições relacionadas a interpretação e implementação dos conceitos associados aos Processos Quânticos e Processos Quânticos Parciais. Esses processos, quando computados no ambiente de execução *VPE-qGM*, requerem menor tempo de execução e resultam em incremento no desempenho, viabilizando a simulação de algoritmos mais complexos.

4.1 Modelo qGM

O modelo *qGM* está fundamentado na teoria dos espaços coerentes, introduzida por Girard (GIRARD, 2004). Os objetos do domínio de processos $\mathbb{D}_\infty$ (REISER; AMARAL; COSTA, 2007a,b) são relevantes no contexto deste trabalho por definirem conjuntos coerentes que interpretam processos quânticos possivelmente infinitos, rotulados por pontos de um espaço geométrico, os quais caracterizam a base computacional para os conjuntos de vetores e matrizes de dimensão n do espaço de Hilbert ($\mathbb{H}$).

A computação é concebida como uma transição de estados associada a uma localização espacial, obtida a partir da sincronização de processos clássicos, caracterizando a unidade de tempo computacional. Com base na representação parcial associada aos objetos do modelo *qGM*, tem-se a possibilidade de obter diferentes descrições e interpretações acerca da evolução do estado de um sistema quântico.

O modelo *qGM* substitui a noção de portas quânticas pelo conceito de sincronizações 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*: Posição de escrita em um espaço de memória global e compartilhada, na qual é armazenado o resultado calculado pelo *PE*.

A estrutura de memória, modelando o vetor de estado do sistema quântico, associa cada posição e correspondente valor armazenado a um estado e amplitude, respectiva-

mente.

Um *PE* pode ler de várias posições de memória (espaço de estados), porém só pode escrever em uma posição. Exemplificando, considera-se a aplicação $H|\psi\rangle$, descrita em (21). Percebe-se a ocorrência de duas operações de atribuição clássicas simultâneas:

- (i) somatório das amplitudes, associando o resultado, acrescido do valor de normalização $\frac{1}{\sqrt{2}}$ ao estado $|0\rangle$;
- (ii) subtração das amplitudes, associando o resultado, acrescido do valor de normalização $\frac{1}{\sqrt{2}}$ ao estado $|1\rangle$.

O modelo *qGM* estabelece que (i) e (ii) podem ser representados por dois *PEs* sincronizados. As correspondentes parametrizações definem um comportamento análogo aos vetores componentes que modelam a matriz associada à correspondente transformação quântica. Dessa forma, durante a simulação, ocorre a execução simultânea dos *PEs*, manipulando os dados presentes no espaço de memória de forma equivalente a aplicação da matriz unitária de *H* sobre o vetor de estado do sistema, simulando o comportamento do sistema quântico.

4.2 Ambiente VPE-qGM

O ambiente *VPE-qGM* está em desenvolvimento com o objetivo de oferecer suporte a modelagem e simulação de algoritmos quânticos considerando as abstrações do modelo *qGM*. Dentre os componentes arquiteturais do ambiente, os seguintes são destacados:

- *qPE* (*Quantum Process Editor*): IDE para o desenvolvimento gráfico dos algoritmos;
- *qME* (*Quantum Memory Editor*): interface para definição do estado inicial do sistema quântico. Um grid de memória armazena cada estado básico e a correspondente, modelando um vetor de estado de forma análoga à descrita na Equação (13);
- *qS* (*Quantum Simulator*): A partir das estruturas definidas nas interfaces do *qPE* e *qME*, o *qS* realiza a simulação do algoritmo quântico, suportando duas abordagens: (i) simulação passo-a-passo, a qual é relevante para viabilizar uma análise detalhada da evolução da simulação; (ii) simulação distribuída (MARON et al., 2010) obtida a partir da integração com o ambiente *VirD-GM* (*Virtual Distributed Geometric Machine*) (FONSECA et al., 2007);
- *qGM-Analyzer*: Biblioteca para realização das computações associadas aos componentes que descrevem cada passo do algoritmo quântico, estando integrada à interface *qS*. O suporte para aceleração por GPU é estabelecida nesta biblioteca.

4.3 Processos Quânticos: Visão Conceitual

Um QP é o resultado da sincronização de *PEs* que descrevem por completo uma transformação quântica. Essa construção está ilustrada na Figura 25. A parametrização dos *PEs* define um comportamento análogo aos vetores componentes que modelam a matriz de definição do operador quântico (MDO) modelado. Durante a simulação, ocorre a execução simultânea dos *PEs*, manipulando dados no espaço de memória de forma análoga à aplicação da matriz unitária de *H* sobre o vetor de estado do sistema, simulando o comportamento do sistema quântico.

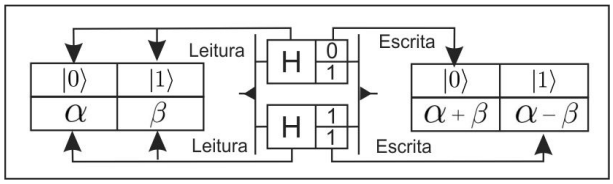


Figura 25: Modelagem da transformação Hadamard a partir de PEs.

A interpretação de $QPPs$ se dá a partir da aplicação parcial de uma transformação quântica devido a existência de indefinições acerca de determinados conjuntos de vetores.

Considere a porta $H^{\otimes 2}$, definida em (22). Cada linha (i) da correspondente matriz em $\mathbb{H}^{\otimes n}$, para $n = 2$, está caracterizada por um PE^i definido em (27). h denota um elemento de $H^{\otimes 2}$, indexado por i (linha) e j (coluna).

$$PE^i|\Phi_t\rangle \equiv |\Phi_{t+1}\rangle := \sum_{j=1}^{2^n} h_{ij}|\Phi_t^j\rangle \tag{27}$$

A sincronização dos PEs^i, para $i \in [0,3]$, modela a computação gerada pela matriz $H^{\otimes 2}$. A execução desse conjunto de *PEs* sobre um estado $|\Phi_0^{01}\rangle$, resultando no estado $|\Phi_1\rangle$, é matricialmente descrito em (28).

$$\begin{aligned} |\Phi_1\rangle &= H \otimes H|\Phi_0^{01}\rangle \\ &= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \end{aligned} \tag{28}$$

Cada conjunto unitário nesta construção interpreta um QPP, correspondendo na notação matricial a uma matriz com apenas uma linha definida, sendo as demais

desconhecidas ($\perp$). Considerando como contexto os elementos da base computacional $(00, 01, 10, 11)$, tem-se a possibilidade de obter-se o estado $|\Phi_1\rangle$ pela união de quatro estados parciais, descritos nas equações a seguir.

$$
\begin{aligned}
|\Phi_1^{00}\rangle_\perp &= H_\perp \otimes H_\perp |\Phi_0\rangle \\
&= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \perp & \perp \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \perp & \perp \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ \perp \\ \perp \\ \perp \end{pmatrix}
\end{aligned}
\tag{29}
$$

$$
\begin{aligned}
|\Phi_1^{01}\rangle_\perp &= H_\perp \otimes H^\perp |\Phi_0\rangle \\
&= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \perp & \perp \end{pmatrix} \otimes \begin{pmatrix} \perp & \perp \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} \perp & \perp & \perp & \perp \\ 1 & -1 & 1 & -1 \\ \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \perp \\ \frac{-1}{2} \\ \perp \\ \perp \end{pmatrix}
\end{aligned}
\tag{30}
$$

$$
\begin{aligned}
|\Phi_1^{10}\rangle_\perp &= H^\perp \otimes H_\perp |\Phi_0\rangle \\
&= \begin{pmatrix} \perp & \perp \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \perp & \perp \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \\ 1 & 1 & -1 & -1 \\ \perp & \perp & \perp & \perp \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \perp \\ \perp \\ \frac{1}{2} \\ \perp \end{pmatrix}
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
|\Phi_1^{11}\rangle_\perp &= H^\perp \otimes H^\perp |\Phi_0\rangle \\
&= \begin{pmatrix} \perp & \perp \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \perp & \perp \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
&= \frac{1}{2} \begin{pmatrix} \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \perp \\ \perp \\ \perp \\ \frac{-1}{2} \end{pmatrix}
\end{aligned} \tag{32}
$$

Logo, o estado $|\Phi_1\rangle$ é aproximado por todos os estados $|\Phi_1^i\rangle_\perp$, com $i \in \{0.0, 0.1, 1.0, 1.1\}$, gerando o estado $|\Phi_1\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$.

Considerando como contexto os valores (0 ou 1) do primeiro *qubit*, tem-se os estados parciais descritos nas equações a seguir.

$$
\begin{aligned}
|\Phi_1^{0.x}\rangle_\perp &= H_\perp \otimes H |\Phi_0\rangle \\
&= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \perp & \perp \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ \frac{-1}{2} \\ \perp \\ \perp \end{pmatrix}
\end{aligned} \tag{33}
$$

$$
\begin{aligned}
|\Phi_1^{1.x}\rangle^\perp &= H_\perp \otimes H |\Phi_0\rangle \\
&= \begin{pmatrix} \perp & \perp \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \perp \\ \perp \\ \frac{1}{2} \\ \frac{-1}{2} \end{pmatrix}
\end{aligned} \tag{34}
$$

Aqui, os estados $|\Phi_1^{0.x}\rangle_\perp$ e $|\Phi_1^{1.x}\rangle^\perp$ são outras aproximações (estados parciais) de $|\Phi_1\rangle$.

Embora não seja o foco deste trabalho, o modelo *qGM* provê interpretações para outras transformações, como as projeções que definem as operações de medidas.

4.4 Implementação

As subseções seguintes descrevem como os conceitos associados aos Processos Quânticos e Processos Quânticos Parciais estão implementados na biblioteca *qGM-Analyzer*.

4.4.1 Transformações Quânticas não Controladas

O QP é o componente capaz de modelar uma transformação quântica em sua totalidade. A Figura 26 ilustra um QP associado a um sistema de 3 *qubits*, sua correspondente composição a partir de *PEs* e a estrutura que modela essa construção na biblioteca *qGM-Analyzer*. As funções U_k são associadas à definição das transformações quânticas de 1 *qubit*. O produto escalar entre os valores retornados por cada função resulta no primeiro elemento de cada tupla armazenada pelo QP_Table. O segundo elemento indica a posição no qual esse elemento é armazenado dentro da correspondente matriz. Tal indexação se faz necessária uma vez que elementos nulos não são armazenados, de forma que a posição real ocupada na matriz não reflete a posição real do elemento.

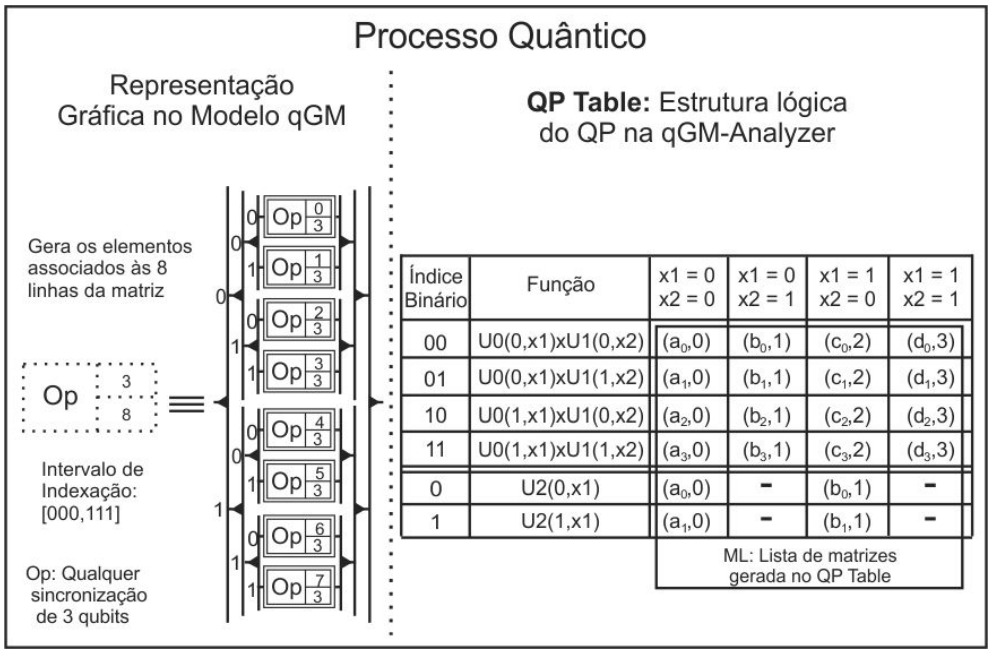


Figura 26: QP e sua correspondente representação por PEs

Analogamente, são descritas sincronizações de quaisquer transformações quânticas **não controladas**, independente da quantidade de *qubits* considerada.

Na Figura 26, ML armazena as matrizes associadas a transformações quânticas, exemplificada por uma transformação quântica de 3 *qubits*. Cada linha da matriz é gerada a partir das funções indicadas na segunda coluna da $QPTable$. Essas funções ($U0$, $U1$ e $U2$) descrevem as transformações quânticas modeladas.

As tuplas de cada linha são obtidas a partir da variação dos valores dos parâmetros $x1$ e $x2$. Cada elemento da tupla corresponde ao valor gerado pelo produto das funções e a correspondente coluna na qual é armazenado.

A ordem de cada matriz em ML é determinada a partir da quantidade de funções agrupadas (n). No caso da Figura 26, tem-se que $n = 2$ para a primeira matriz de ML, denotada por M_1. Analogamente para a segunda matriz, indicada por M_2, tem-se $n =$

1. Destaca-se que a ordem de cada matriz em ML pode ser determinada de forma arbitrária, porém para n muito grande, tem-se um aumento exponencial do consumo de memória. Dessa forma, um balanceamento entre a ordem e a quantidade de matrizes em ML ($|ML|$) influencia diretamente no desempenho da simulação.

Faz-se necessário, além da ML, a geração de uma estrutura de lista (veja em (35)) contendo valores auxiliares para indexação das amplitudes do espaço de estados a serem multiplicadas por cada valor das matrizes em ML. Nessa lista, q indica a quantidade total de *qubits* do sistema quântico.

$$sizesList = [2^{q-n}, 2^{q-(2*n)}, ..., 2^{q-(|ML|*n)}] \tag{35}$$

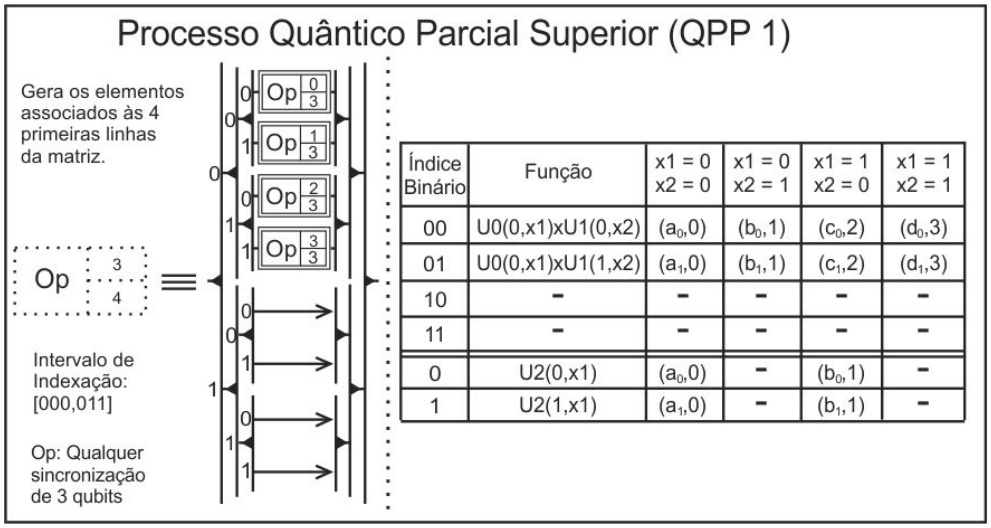
Cada linha de uma matriz possui uma indexação binária de n *bits*. Exemplificando, na Figura 26, a indexação 000 seleciona a primeira linha de cada matriz (m_{00} da primeira matriz e m_0 da segunda matriz), viabilizando a obtenção da amplitude do primeiro estado base do sistema ($|000\rangle$) a partir da expressão recursiva definida por (36), onde:

- $|ML|$ é a quantidade de matrizes em ML;
- P é uma posição base (inicializada com $P = 0$) para indexação da amplitude de um dos estados da base computacional;
- m indica uma matriz em ML (inicializado em $m = 1$);
- l_m indexa a linha l da matriz m;
- n_m é a ordem da matriz m;
- SL armazena a $SizesList$;
- T' é a tupla indexada por $ML_{m,l_m,c_{l_m}}$;
- T'' é a tupla indexada por $ML_{|ML|,l_{|ML|},c}$;
- k é a amplitude de um estado da base computacional.

$$\begin{aligned} F(P,m) &= \sum_{c=1}^{2^{nm}} T'_0 \times F((P + (SL_m \times T'_1)), m+1) \\ F(P,|ML|) &= \sum_{c=1}^{2^{n|ML|}} T''_0 \times k_{P+(T''_1)} \end{aligned} \tag{36}$$

De acordo com as especificações do modelo *qGM*, um QP pode, ainda, ser representado por uma sincronização de vários $QPPs$. Nessa concepção, esta representação viabiliza o particionamento do QP da Figura 26 em dois $QPPs$.

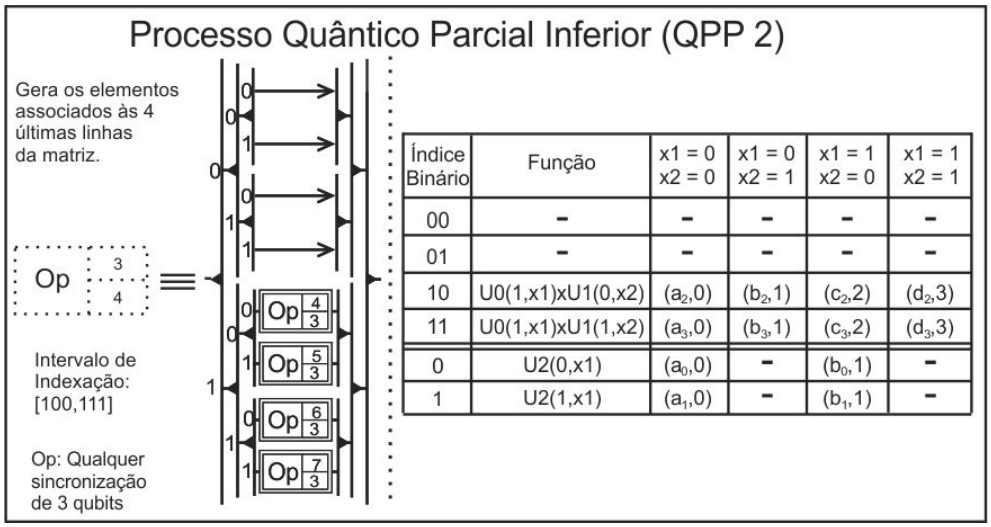
Na Figura 27, tem-se o $QPP\ 1$ que é responsável por gerar todas as novas amplitudes dos estados da base computacional contidos no intervalo $[0, 3]$, sendo representado pelo componente gráfico tracejado na extremidade esquerda.



Índice Binário	Função	x1 = 0 x2 = 0	x1 = 0 x2 = 1	x1 = 1 x2 = 0	x1 = 1 x2 = 1
00	U0(0,x1)xU1(0,x2)	(a_0,0)	(b_0,1)	(c_0,2)	(d_0,3)
01	U0(0,x1)xU1(1,x2)	(a_1,0)	(b_1,1)	(c_1,2)	(d_1,3)
10	-	-	-	-	-
11	-	-	-	-	-
0	U2(0,x1)	(a_0,0)	-	(b_0,1)	-
1	U2(1,x1)	(a_1,0)	-	(b_1,1)	-

Figura 27: QPP que gera as amplitudes no intervalo $[0, 3]$

Analogamente, tem-se na extremidade direita da Figura 28 o componente gráfico tracejado associado ao $QPP\ 2.$ o qual gera as amplitudes dos estados da base computacional no intervalo $[4, 7]$, independentemente da execução do $QPP\ 1.$



Índice Binário	Função	x1 = 0 x2 = 0	x1 = 0 x2 = 1	x1 = 1 x2 = 0	x1 = 1 x2 = 1
00	-	-	-	-	-
01	-	-	-	-	-
10	U0(1,x1)xU1(0,x2)	(a_2,0)	(b_2,1)	(c_2,2)	(d_2,3)
11	U0(1,x1)xU1(1,x2)	(a_3,0)	(b_3,1)	(c_3,2)	(d_3,3)
0	U2(0,x1)	(a_0,0)	-	(b_0,1)	-
1	U2(1,x1)	(a_1,0)	-	(b_1,1)	-

Figura 28: QPP que gera as amplitudes no intervalo $[4, 7]$

A utilização de $QPPs$ contribui com a possibilidade de estabelecer interpretações parciais de uma transformação quântica, permitindo que alguns parâmetros/processos sejam desconhecidos, uma vez que, em um **contexto local** da computação de cada *QPP*, é possível gerar apenas conjuntos restritos de elementos associados a um MDO. *QPPs* complementares (que interpretam conjuntos disjuntos de linhas) podem ser sincronizados e executados de forma independente (localmente ou em diferentes nodos de

processamento). Quanto mais *QPPs* forem sincronizados, menor é a porção da transformação quântica a ser interpretada por cada um, caracterizando um menor custo de execução.

4.4.2 Definição de Transformações Quânticas Controladas

Para transformações quânticas **não controladas**, é possível modelar toda a evolução do estado global do sistema quântico a partir de **apenas um** QP. Entretanto, essa possibilidade não é valida para transformações quânticas controladas. A principal diferença pode ser vista na Figura 29, na qual as seguintes condições são descritas:

- Para geração da transformação $H^{\otimes 2}$, a expressão (H, H) se mantém para todos os vetores componentes, variando apenas as correspondentes parametrizações;
- Para a descrição da transformação *CNOT*, diferentes expressões são necessárias. Essa diferença se justifica pela interpretação intrínseca da transformação *CNOT*.[1]

Essa interpretação pode ser estendida para qualquer outra transformação quântica, com dois *qubits* ou mais.

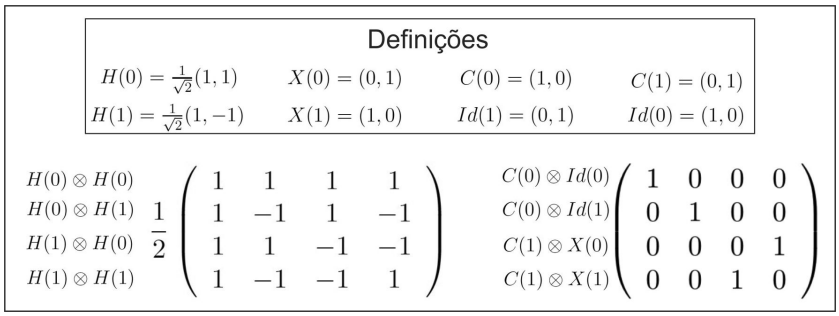


Figura 29: Parametrização das transformações H e CNOT.

A completa descrição da transformação *CNOT* se dá a partir das expressões definidas em (37), que originam um conjunto de $QPPs$, denominado *QPP_Set*. Os $QPPs$ deste conjunto, para a transformação *CNOT*, têm suas estruturas ilustradas na Figura 30. O QPP_1, representado na Figura 30(a) e associado à Exp_1, descreve a evolução dos estados quânticos nos quais o estado do *qubit* de controle é 1 (exigindo a aplicação da transformação *X* no *qubit* alvo). A evolução dos estados nos quais o *qubit* de controle é 0 é modelada por Exp_2, a qual gera o QPP_2, ilustrado na Figura 30(b). Como esses estados não são modificados, o QPP_2 torna-se passível de ser executado.

$$Exp_1 = C(1), X \tag{37}$$
$$Exp_2 = C(0), Id$$

[1] *Se o estado do primeiro qubit for 0, não faça nada; se o estado do primeiro qubit for 1, aplique a transformação X no segundo qubit, mantendo o primeiro inalterado.*

Índice Binário	Função	x1 = 0 x2 = 0	x1 = 0 x2 = 1	x1 = 1 x2 = 0	x1 = 1 x2 = 1
00	-	-	-	-	-
01	-	-	-	-	-
10	C(1,x1) x X(0,x2)	-	-	-	(1,3)
11	C(1,x1) x X(1,x2)	-	-	(1,2)	-

(a) QPP_1: altera amplitudes

Índice Binário	Função	x1 = 0 x2 = 0	x1 = 0 x2 = 1	x1 = 1 x2 = 0	x1 = 1 x2 = 1
00	C(0,x1) x Id(0,x2)	(1,0)	-	-	-
01	C(0,x1) x Id(1,x2)	-	(1,1)	-	-
10	-	-	-	-	-
11	-	-	-	-	-

(b) QPP_2: não altera amplitudes

Figura 30: QPPs para modelagem da transformação CNOT

Genericamente, tem-se que $|QPP\_Set| = |Exp| = 2^{nC}$, sendo nC a quantidade total de *qubits* de controle de todas as transformações quânticas modeladas. Entretanto, é necessário criar/executar apenas os $QPPs$ pertencentes a um subconjunto (*QPP_Subset*) de *QPP_Set*. Se considerada apenas uma transformação controlada, tem-se que $|QPP\_Subset| = 1$. Quando consideradas sincronizações de transformações controladas, tem-se $|QPP\_Subset| = 2^{nC} - 1$.

Considere, agora, a aplicação de duas transformações *CNOT*, conforme ilustrado na Figura 31. Nessa configuração, tem-se as seguintes possibilidades:

- $Q_1 = |1\rangle$ e $Q_3 = |1\rangle$: Aplica-se *X* à Q_2 e Q_4;
- $Q_1 = |1\rangle$ e $Q_3 = |0\rangle$: Aplica-se *X* à Q_2 e *Id* à Q_4;
- $Q_1 = |0\rangle$ e $Q_3 = |1\rangle$: Aplica-se *X* à Q_4 e *Id* à Q_2;
- $Q_1 = |0\rangle$ e $Q_3 = |0\rangle$: Aplica-se *Id* à Q_2 e Q_4.

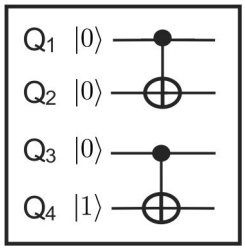


Figura 31: Sincronização de transformações CNOT

No ambiente *VPE-qGM*, a modelagem dessa configuração se dá a partir do conjunto de expressões em (38). Dessa forma, tem-se que $|QPP\_Set| = 4$. Entretanto, o QPP_4,

associado à expressão Exp_4, não realiza nenhuma alteração no sistema, tornando-se passível de criação/execução.

$$
\begin{aligned}
Exp_1 &= C(1), X, C(1), X \\
Exp_2 &= C(1), X, C(0), Id \\
Exp_3 &= C(0), Id, C(1), X \\
Exp_4 &= C(0), Id, C(0), Id
\end{aligned}
\tag{38}
$$

Quando da sincronização de transformações controladas com transformações não controladas (diferente de *Id*), todas as amplitudes são modificadas. Dessa forma, tem-se que $QPP\_Subset = QPP\_Set$. A configuração ilustrada na Figura 32 é modelada a partir das expressões definidas em (39). Dessa forma, dois $QPPs$, identificados por QPP_1 e QPP_2, respectivamente associados às expressões em Exp_1 e Exp_2, são necessários. Entretanto, não é possível descartar a execução de QPP_2, uma vez que este, neste caso, realizam-se alterações nas amplitudes de alguns estados. Essas alterações são decorrentes da transformação *H*, a qual **sempre** é aplicada ao último *qubit*, independente do estado de controle da transformação *CNOT*.

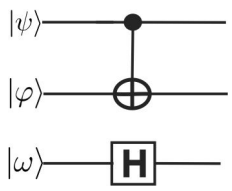


Figura 32: Sincronização de transformação CNOT com H

$$
\begin{aligned}
Exp_1 &= C(1), X, H \\
Exp_2 &= C(0), Id, H
\end{aligned}
\tag{39}
$$

4.4.3 Função Recursiva

Após a criação dos $QPPs$, tem-se a aplicação de um operador recursivo que atua sobre as matrizes para cálculo das amplitudes no novo estado do sistema quântico. Essa função realiza uma combinação entre diferentes tuplas para geração dos valores correspondentes à MDO aplicada. Além disso, é gerado um valor que indexa a amplitude do estado atual do sistema que deve ser multiplicada pelo valor gerado. A descrição algorítmica desse procedimento, com algumas otimizações, pode ser vista na Figura 33.

O custo de execução deste algoritmo cresce exponencialmente com o aumento na quantidade de *qubits* que compõem o sistema quântico modelado. Apesar da elevada complexidade temporal, a complexidade espacial é baixa, uma vez que apenas valores

```
if matrixIndex = numMatrices − 1 then
    for l = 0 to size(Matrices[matrixIndex]) do
        res ← 0;
        line ← Matrices[matrixIndex][l][0];
        linePos ← Matrices[matrixIndex][l][0];
        for column = 0 to size(line) do
            pos ← basePos + line[column][1];
            res ← res + (partialValue × line[column][0] × memory[1][pos]);
        end for
        writePos ← memPos + (linePos × sizesList[matrixIndex]);
        res ← res + memory[0][writePos];
        memory[0][writePos] ← res;
    end for
else
    for l = 0 to size(Matrices[matrixIndex]) do
        line ← Matrices[matrixIndex][l][1];
        linePos ← Matrices[matrixIndex][l][0];
        for column = 0 to size(line) do
            next_basePos ← basePos + (line[column][1] × sizesList[matrixIndex]);
            next_partialValue ← partialValue × line[column][0];
            ApplyValues(Matrices, numMatrices, sizesList, memory,      next_partialValue, matrixIndex    +
            1, next_basePos, memPos + (linePos × sizesList[matrixIndex]));
        end for
    end for
end if
```

Figura 33: Algoritmo estruturado para execução das computações sobre QPs e QPPs

temporários são armazenados.

4.5 Considerações Finais

A simulação quântica a partir do ambiente *VPE-qGM* vêm sendo aprimorada ao longo de vários projetos no qual está inserido. A utilização de interfaces gráficas para modelagem e simulação dos algoritmos tem por objetivo simplificar o processo de desenvolvimento e interpretação dos resultados.

As limitações decorrentes da simulação por processos elementares foram superadas pelas implementações para suporte à Processos Quânticos. A possibilidade de geração parcial de estados quânticos com diferentes níveis de granulosidade permite o controle sobre a computação a ser executada. Embora não seja o principal foco deste trabalho, este controle permite especificar a distribuição de tarefas quando considerada a simulação quântica distribuída a partir do ambiente *VirD-GM*, conforme apresentado em (AVILA et al., 2012).

A utilização de Processos Quânticos Parciais para descrição de transformações controladas permite a identificação dos subprocessos que realizam alterações no espaço de estados, executando-os e ignorando demais subprocessos que não contribuem para o novo estado do sistema quântico. Dessa forma, o custo computacional dessas operações pode ser reduzido gerando uma melhora no desempenho da simulação.

A função recursiva que percorre as estruturas de Processos Quânticos e Processos Quânticos Parciais foi definida de forma a gerar dinamicamente todos os valores de definem a transformação quântica. Dessa forma tem-se uma redução no consumo de

memória, uma vez que os valores usualmente armazenados na matriz resultante do produto tensorial são descartados logo após serem utilizados para computar parcialmente uma amplitude do novo estado quântico. Ainda mais, são evitadas computações redundantes que usualmente ocorrem na operação de produto tensorial, seja ele executado de forma distribuída ou não.

Os resultados obtidos por essa forma de simulação sequencial podem ser vistos na Seção 6.1.

5 EXECUÇÃO PARALELA DE PROCESSOS QUÂNTICOS

A simulação sequencial de algoritmos quânticos, mesmo que considerando técnicas de otimização altamente eficientes, mostra-se insuficiente quando considerados algoritmos quânticos com mais de 30 *qubits*, como pode ser visto a partir dos resultados descritos em (VIAMONTES, 2007).

Nos cenários nos quais a expansão exponencial do uso de memória pode ser controlado, o tempo de simulação passa a ser o problema dominante, uma vez que a quantidade de operações a serem realizadas para se obter o novo estado do sistema quântico também cresce exponencialmente. Tendo em vista este segundo problema, este Capítulo introduz os esforços iniciais que viabilizam a execução paralela de Processos Quânticos no ambiente *VPE-qGM*.

5.1 Estruturas de Dados

Seguindo as especificações previamente discutidas acerca da construção de Processos Quânticos, este pode ser definido a partir da construção de matrizes básicas de menor ordem, as quais são combinadas a partir de uma função iterativa de forma a gerar dinamicamente os elementos correspondentes à matriz resultante do produto tensorial.

Para a execução de um QP, a biblioteca de simulação faz uso de vários parâmetros, conforme ilustrado na Figura 34, a qual exemplifica um estudo de caso baseado em uma transformação $H^{\otimes 2} \otimes CNOT \otimes H^{\otimes 4}$.

Seguem as descrições dos parâmetros considerados, os quais são armazenados em um vetor gerado pela biblioteca *NumPy*:

1. Lista de Matrizes (*matrices*): matrizes básicas geradas pelo *host-code*, não armazenando valores nulos;

2. Lista de Posições (*positions*): posição de um elemento na correspondente matriz, sendo utilizado pelo *kernel* de execução para identificar a amplitude do vetor de estado a ser acessado durante a simulação;

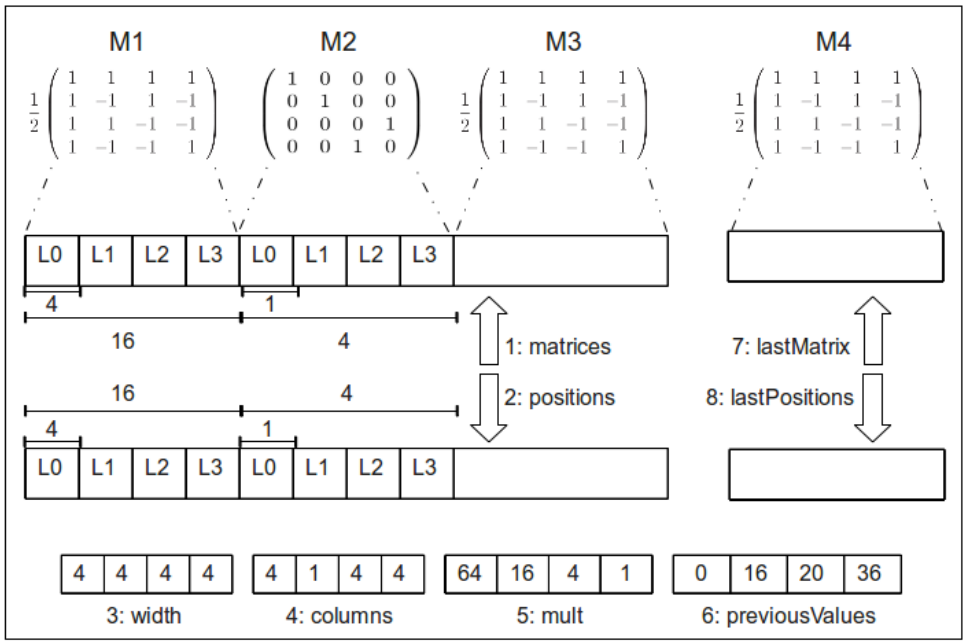


Figura 34: Organização dos dados para a transformação $H^{\otimes 2} \otimes CNOT \otimes H^{\otimes 4}$

3. Colunas Originais das Matrizes (*width*): quantidade de colunas na definição original de cada matriz;

4. Colunas Otimizadas das Matrizes (*columns*): quantidade de colunas com elementos não nulos armazenados;

5. Multiplicadores (*mult*): valores auxiliares para indexação das amplitudes do vetor de estado;

6. Deslocamento (*previousValues*): Quantidade de elementos armazenados correspondentes à matrizes anteriores;

7. Última matriz de dados (*lastMatrix*): armazenada em separado das demais para reduzir a quantidade de cálculos exigidos para indexação dos seus elementos. Como esses dados são constantemente acessados, operações complexas sobre seus valores devem ser evitados;

8. Ultima matriz de posições (*lastPositions*): também possui armazenamento em uma estrutura exclusiva para simplificar a indexação de elementos constantemente acessados.

Essas estruturas de dados são utilizadas uma vez que todos os valores correspondentes às matrizes são armazenados na forma de um vetor contíguo de dados. As informações acerca dos tamanhos das matrizes indicam quais dados pertencem a quais matrizes e sua respectiva posição dentro da matriz. A forma de combinação desses parâmetros pode ser vista em detalhes nos trechos do *CUDA kernel* apresentados na Seção 5.3.

A opção por essa forma de estruturação se justifica pela futura necessidade de extensão dessas construções para suporte à Processos Quânticos Parciais, na qual serão utilizados blocos de dados e de *threads* bidimensionais. Atualmente, por se tratar de uma simulação considerando apenas um Processo Quântico, blocos unidimensionais são utilizados.

5.2 Alocação de Dados na GPU

A disposição dos dados ao longo dos diferentes espaços de memória da *GPU* tem especial importância para garantir um bom desempenho da simulação. Os parâmetros de definição dos Processos Quânticos não se modificam durante a simulação e são comuns a todas as *threads*. Dessa forma, a alocação desses dados no espaço de memória constante torna-se uma opção adequada. As cópias de dados são realizadas a partir de chamadas realizadas pela biblioteca *PyCUDA* de acordo com os seguintes passos:

1. No *CUDA kernel*, os dados a serem armazenados na memória constante são declarados com a seguinte sintaxe: $\_\_device\_\_ \; \_\_constant\_\_ \; dType \; dadosGPU[tamanho]$;

2. O endereço de memória na *GPU* referente à *'dadosGPU'* é obtido no *host-code* pela seguinte chamada: $endGPU = kernel.get\_global('dadosGPU')[0]$;

3. A cópia de dados também é executada a partir do *host-code*. Os dados da memória do *host* são transferidos para o espaço de memória indexado por *'endGPU'*, referente a localização do conteúdo armazenado por *'dadosCPU'*. Tal processo se dá a partir da seguinte chamada da biblioteca *PyCUDA*: $pycuda.driver.memcpy\_htod(endGPU, objetoNumPy)$. Destaca-se que os dados copiados devem estar armazenados na memória RAM através de um objeto da biblioteca *NumPy*.

Além dos dados referentes a parametrização dos Processos Quânticos, dois vetores de estado são encontrados no espaço de memória do *host*: um modelando o estado atual do sistema quântico (*readMemory*) e outro modelando o próximo estado (*writeMemory*), o qual é calculado em paralelo por várias *threads*. Antes da execução do *CUDA kernel* para início da computação, ambos vetores são copiados para a memória global da *GPU* a partir dos seguintes métodos:

1. $readMemory = numpy.array(numpy.zeros(2^q), dtype = numpy.complex64, order =' C')$ corresponde a criação do vetor de estado atual do sistema. Inicialmente todos os valores são iguais a zero, porém os valores podem ser modelados de acordo com as exigências do algoritmo a ser simulado;

2. $readMemory\_gpu = gpuarray.to\_gpu(readMemory)$ representa a cópia do vetor de estado atual para a memória global da *GPU*;

3. $writeMemory\_gpu = gpuarray.zeros(2^q, dtype = numpy.complex64, order =' C')$ aloca o espaço para o novo vetor de estado na memória global da *GPU*, inicializando seu conteúdo com zeros.

5.3 CUDA Kernel

O *CUDA kernel* é uma adaptação do algoritmo recursivo apresentado na Seção 4.4.3, apresentando o mesmo comportamento porém implementado de forma iterativa. Essa adaptação se faz necessária uma vez que a maior parte das *GPUs* não oferece suporte à recursão. Como o *kernel* aqui descrito tem um comportamento similar ao *Kronecker Product*, ele é capaz de atuar sobre um número arbitrário de matrizes básicas. Cada *CUDA thread* possui sua pilha de execução interna e controles de execução para definir os limites de acesso em cada matriz.

A computação de cada *thread* pode ser dividida em sete passos, descritos a seguir.

Passo 1: **Inicialização** dos parâmetros na memória constante, os quais são comuns a todas às *CUDA threads* da aplicação. $TOTAL\_ELEMENTS$, $LAST\_MATRIX\_ELEMENTS$ e $STACK\_SIZE$ são definidos em tempo de execução pelo interpretador do *PyCUDA*.

$$\_\_device\_\_ \ \_\_constant\_\_ \ cuFloatComplex \ matricesC[TOTAL\_ELEMENTS];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ positionsC[TOTAL\_ELEMENTS];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ cuFloatComplex \ lastMatrixC[LAST\_MATRIX\_ELEMENTS];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ lastPositionsC[LAST\_MATRIX\_ELEMENTS];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ widthC[STACK\_SIZE + 1];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ multC[STACK\_SIZE + 1];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ columnsC[STACK\_SIZE + 1];$$
$$\_\_device\_\_ \ \_\_constant\_\_ \ int \ previousElementsC[STACK\_SIZE + 1];$$

Passo 2: **Alocação de dados na memória compartilhada** e correspondente inicialização são ambos executados por todas as *threads* de um mesmo bloco. *SHARED_SIZE* é definido em tempo de execução, porém geralmente assume o valor $blockDimx.x \times 4$.

$$\begin{aligned}
&__shared__ \ cuFloatComplex \ newAmplitudes[SHARED_SIZE];\\
&for \ (int \ i = 0; \ i < 4; \ i++)\{\\
&\quad ind = threadIdx.x \times 4 + i;\\
&\quad newAmplitudes[ind] = make_cuFloatComplex(0, 0); \}\\
&__syncthreads();
\end{aligned}$$

Passo 3: **Definição dos limites de acesso** dentro de cada matriz, determinando quais elementos cada *CUDA thread* pode acessar dependendo do seu *id* e bloco na qual está inserida. Os vetores *begin*, *count* e *end* são locais a cada *thread* e auxiliam

no controle e indexação dos elementos de cada matriz em *matricesC* e *positionsC*. A operação $(thId\&(widthC[c]-1))$ é análoga à operação de módulo $thId\%widthC[c]$, porém executada como um *and* bit-a-bit que possui melhor desempenho na *GPU*.

$$
\begin{aligned}
&int\ thId = blockDim.x \times blockIdx.x + threadIdx.x;\\
&for(int\ c = STACK\_SIZE - 1;\ c > -1;\ c--)\{\\
&\quad begin[c] = (thId\&(widthC[c]-1)) \times columnsC[c];\\
&\quad count[c] = begin[c];\\
&\quad end[c] = begin[c] + columnsC[c];\\
&\quad thId = floorf((thId/widthC[c]));\}
\end{aligned}
$$

Passo 4: **Avanço nas matrizes**, sendo análogo a um passo recursivo provendo as multiplicações parciais entre os elementos indexados de acordo com os valores indicados por *count*.

$$
\begin{aligned}
&while\ (ind < STACK\_SIZE)\{\\
&\quad position = positionsC[previousElementsC[ind]];\\
&\quad mod = (position + count[ind])\&(widthC[ind]-1);\\
&\quad readStack[top+1] = readStack[top] + (mod \times multC[ind]);\\
&\quad offset = previousElementsC[ind];\\
&\quad element = matricesC[offset + count[ind]];\\
&\quad valueStack[top+1] = cuCmulf(valueStack[top], element);\\
&\quad top++;\\
&\quad ind++;\}
\end{aligned}
$$

Passo 5: **Escrita na memória compartilhada**, correspondente a uma atualização

parcial de 4 amplitudes do novo vetor de estado do sistema.

$$
\begin{aligned}
&c = 0;\\
&for(int\ j = 0;\ j < 4;\ j++)\{\\
&\quad res = make\_cuFloatComplex(0, 0);\\
&\quad for(int\ k = 0;\ k < columnsC[STACK\_SIZE];\ k++)\{\\
&\quad\quad mod = (lastPositionsC[c]\&(widthC[ind] - 1))\\
&\quad\quad readPos = readStack[top] + mod;\\
&\quad\quad mult = cuCmulf(lastMatrixC[c], readMemory[readPos]);\\
&\quad\quad res = cuCaddf(res, mult);\\
&\quad\quad c++;\}\\
&\quad writePos = threadIdx.x \times 4 + j;\\
&\quad partAmp1 = newAmplitudes[writePos];\\
&\quad partAmp2 = cuCmulf(res, valueStack[top]);\\
&\quad newAmplitudes[writePos] = cuCaddf(partAmp1, partAmp2);\}
\end{aligned}
$$

Passo 6: **Atualização dos índices** para iteração ao longo dos valores em cada matriz. Esse processo ocorre até que todos os índices apontem para o último elemento de cada linha em todas as matrizes.

$$
\begin{aligned}
&top--;\\
&ind--;\\
&count[ind]++;\\
&while(count[ind] == end[ind]\{\\
&\quad count[ind] = begin[ind];\\
&\quad ind--;\\
&\quad count[ind]++;\\
&\quad top--;\}
\end{aligned}
$$

Passo 7: **Cópia de dados** da memória compartilhada para a memória global da *GPU*, realizada após todas as *threads* de um mesmo bloco terem finalizados suas computações.

$$
\begin{aligned}
&thId = blockDim.x \times blockIdx.x + threadIdx.x;\\
&for\ (int\ i = 0;\ i < 4;\ i++)\{\\
&\quad writePos = thId \times 4 + i;\\
&\quad readPos = threadIdx.x \times 4 + i;\\
&\quad writeMemory[writePos] = newAmplitudes[readPos];
\end{aligned}
$$

Todos os passos descritos compõem o *CUDA kernel* e definem a computação de

uma *thread*, recalculando 4 amplitudes do novo vetor de estado de acordo com as transformações quânticas aplicadas.

5.4 Considerações Finais

O desenvolvimento da extensão para suporte a simulação quântica a partir de *GPUs* no ambiente *VPE-qGM* constitui uma solução para redução do tempo de simulação. As etapas de estudo de desenvolvimento conduzidas neste trabalho contemplam os esforços iniciais que visam a estruturação e superação dos primeiros obstáculos associados à programação no paradigma *GPGPU*.

O suporte às configurações mais elementares de transformações quânticas foi estabelecido, viabilizando a simulação das transformações quânticas não controladas descritas na Seção 2.1.6. A utilização na biblioteca *PyCUDA* é responsável por simplificar o processo de desenvolvimento, oferecendo métodos simples e rotinas para cópias de dados e geração dinâmica de código.

O *CUDA kernel*, implementado na linguagem *C* como uma *string* dentro do *host-code* em *Python*, segue uma abordagem de execução iterativa, porém com comportamento similar à versão recursiva descrita na Seção 4.4.3. Os resultados desses esforços podem ser vistos na Seção 6.2, na qual são apresentados os principais resultados acerca da simulação paralela.

É importante salientar que, devido ao estágio inicial de desenvolvimento, apenas algumas das políticas de otimização para programação em *GPU* descritas na Seção 3.5 foram implementadas. Entretanto, através de algumas reestruturações pouco complexas é possível manipular o comportamento das *threads* para que outras condições necessárias para uma execução mais eficiente sejam atendidas.

6 RESULTADOS

Neste capítulo são apresentados os principais resultados obtidos a partir das contribuições geradas por este trabalho. Os resultados estão caracterizados de duas formas:

- Resultados para simulação sequencial, no qual várias configurações de transformações quânticas são simuladas a partir de Processos Quânticos e Processos Quânticos Parciais;
- Resultados para simulação paralela sobre *GPUs*, no qual tem-se os resultados associados à simulação de transformações quânticas não controladas.

6.1 Simulação Sequencial

Para validação e análise de desempenho da simulação de algoritmos quânticos a partir de QPs no *VPE-qGM*, dois estudos de caso foram considerados. O primeiro contempla *benchmarks*, compostos fundamentalmente por transformações controladas, selecionados dentre os disponíveis em (D. MASLOV; SCOTT, 2011). Essa escolha é justificada por dois pontos principais:

- Disponibilização de código fonte para geração de todos os algoritmos;
- Descrição de algoritmos quânticos com muitos *qubits* e dezenas/centenas de operadores.

O segundo considera transformações quânticas *Hadamard* de até 14 *qubits* ($H^{\otimes 11}$, $H^{\otimes 12}$, $H^{\otimes 13}$ e $H^{\otimes 14}$), validando, assim, as otimizações para operadores quânticos não controlados.

A metodologia dos testes utilizada contempla, para cada estudo de caso, a realização de 10 simulações. A máquina utilizada na simulação possui as seguintes características principais: processador Intel Core i5-2410M @ 2.3 GHz, 4GB RAM e sistema operacional Ubuntu 11.10 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 versão anterior

da biblioteca *qGM-Analyzer*, a qual suporta a simulação de algoritmos quânticos a partir de *PEs*, considerando as descrições introduzidas em (MARON et al., 2011). As principais características de cada algoritmo, juntamente com resultados obtidos pelas correspondentes simulações, são sumarizados na Tabela 4.

Tabela 4: Algoritmos quânticos simulados

Algoritmo	Qubits	Transformações	Simulação c/ QPs		Simulação c/ PEs	
			Tempo (s)	Mem (MB)	Tempo (s)	Mem (MB)
9symd2	12	28	0,400	12	38,805	12
$gf2^4$	12	19	0,236	12	26,352	12
$gf2^5$	15	29	1,328	13	372,488	13
$gf2^6$	18	41	11,264	24	5081,553	22
$gf2^7$	21	55	135,013	92	NS[1]	NS
$gf2^8$	24	85	1532,015	524	NS	NS
ham15_1	15	132	6,872	13	1778,170	13
ham15_2	15	70	4,708	13	925,319	13
ham15_3	15	109	8,436	13	1400,632	13
mod1024adder	20	55	44,099	60	NS	NS
rc_adder	16	19	2,358	15	549,079	14
$H^{\otimes 11}$	11	11	6,816	12	7,882	12
$H^{\otimes 12}$	12	12	25,292	12	28,281	12
$H^{\otimes 13}$	13	13	97,401	12	111,572	12
$H^{\otimes 14}$	14	14	348,923	12	496,934	12

NS: Não Suportado. Tempo de simulação acima de 4 horas.

Como pode ser observado nos resultados descritos na Tabela 4, algoritmos quânticos com até 24 *qubits* foram simulados. O consumo de memória se mantém similar para ambas as bibliotecas. Como as otimizações propiciadas pelos QPs contemplam apenas as transformações quânticas que modelam o algoritmo, a maior parte do consumo de memória se dá pela criação dos vetores que armazenam as amplitudes dos estados.

Os tempos de simulação da biblioteca *qGM-Analyzer* com suporte à execução de $QPPs$ apresentaram significante redução quando comparados com a execução por *PEs*.

6.1.1 Análise para Transformações Controladas

Como apresentado na Figura 35, a simulação de transformações quânticas controladas descritas a partir de $QPPs$ apresentaram uma melhora de aproximadamente 99% frente a representação por *PEs*.

O ganho de desempenho obtido é justificado, principalmente, pela otimização voltada a identificação dos $QPPs$ que realizam alterações nas amplitudes do espaço de estados.

Dessa forma, tem-se a execução de apenas um subconjunto do total de $QPPs$ que modelam a transformação quântica, ou seja, apenas parte da transformação quântica é executada. Exemplificando, para cada um dos 41 passos que compõem o algoritmo $gf2^6$, tem-se a geração e execução de operações correspondentes a apenas 2 vetores componentes em uma matriz de ordem 2^{18}. Consequentemente, apenas um subconjunto do total de amplitudes que modelam o estado do sistema quântico é alterado, exigindo uma quantidade menor de operações.

Já na modelagem do mesmo circuito a partir de *PEs*, a cada passo da simulação ocorre a execução de 2^{18} *PEs*. Nessa abordagem, todas as amplitudes do espaço de estados são recalculadas, mesmo que nenhuma alteração nos valores seja efetuada. Devido ao crescimento exponencial de *PEs*, a simulação de algoritmos com mais de 18 *qubits* torna-se inviável a partir dessa forma de representação.

Os algoritmos gf^7, gf^8 e $mod1024adder$ não foram incluídos na Figura 35 por não serem suportados pela modelagem utilizando *PEs*, uma vez que os tempos de execução, nesses casos, foram superiores a 4 horas. Os tempos médios de simulação estão relacionados na Tabela 4

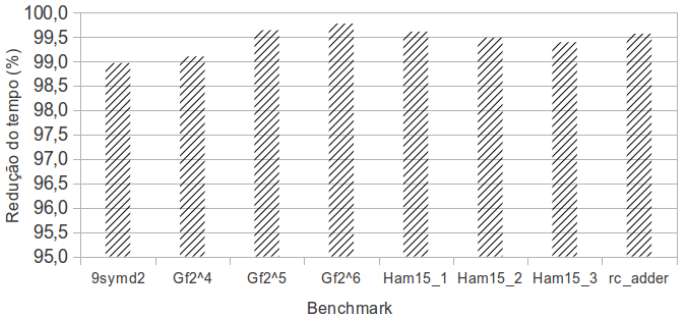


Figura 35: Percentual de melhora para a simulação de transformações controladas a partir de QPPs.

6.1.2 Análise para Transformações não Controladas

O percentual no ganho de desempenho quando da comparação entre a simulação de transformações *Hadamard* a partir de QPs e *PEs* é ilustrada na Figura 36.

Nesses casos, não foi obtido um ganho proporcional aos *benchmarks* constituídos de transformações controladas, uma vez que, nesse caso, todas as amplitudes do espaço de estados são modificadas. Assim, ambas formas de representação resultam na geração e operação sobre os elementos associados aos 2^q vetores componentes da MDO. Entretanto, a redução de até 29% no tempo de simulação se justifica pela geração de todos os elementos a partir de um único QP, enquanto que na modelagem por *PEs*, tem-se a execução de 2^q componentes distintos, resultando em um número maior de operações.

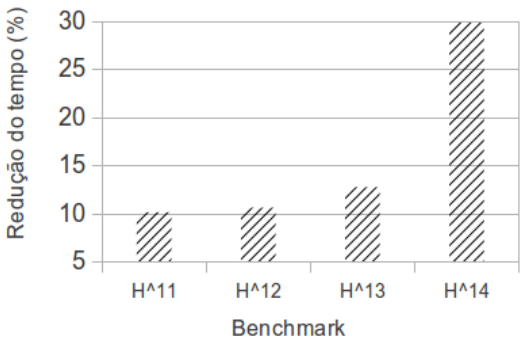


Figura 36: Percentual de melhora para a simulação de transformações Hadamard a partir de QPs.

6.1.3 Relação de Desempenho: VPE-qGM $\times$ Demais Simuladores

Relacionando os resultados obtidos neste trabalho com o estado-da-arte, é possível realizar uma comparação aproximada do desempenho do *VPE-qGM*. Testes realizados em (VIAMONTES, 2007) contemplam a simulação dos algoritmos $ham15\_1$, $ham15\_2$, $ham15\_3$, rc_adder e $9symd2$. Destaca-se que os resultados foram obtidos a partir de uma configuração de *hardware* (processador Athlon 1.2 GHz, 1GB RAM e sistema operacional Linux) diferente da considerada neste trabalho.

Os tempos e consumo de memória obtidos pelo simulador *QuIDDPro*, descritos em (VIAMONTES, 2007), são menores que os obtidos pela simulação a partir de QPs no *VPE-qGM*. Tal resultado pode ser justificado a partir de duas características do ambiente *VPE-qGM*. A primeira está relacionada com a linguagem *Python*, a qual é interpretada e, consequentemente, mais lenta que a linguagem C, na qual o *QuIDDPro* é desenvolvido. A segunda consiste na ausência de otimizações para o armazenamento do espaço de estados dos sistemas quânticos no *VPE-qGM*.

Quando comparado com o simulador *QCSim*, entretanto, a estruturação do *VPE-qGM* garante resultados bem mais significativos, tanto no consumo de memória, quanto no tempo de simulação. Isso ocorre uma vez que o *QCSim* realiza a aplicação de transformações quânticas explicitamente a partir do produto matriz-vetor. Dessa forma, além de não explorar as redundâncias presentes nas estruturas das transformações quânticas, sistemas com mais de 16 *qubits* ultrapassam os 2 GB RAM, inviabilizando sua simulação.

No estágio atual, o consumo de memória do ambiente *VPE-qGM* é influenciado, principalmente, pela estrutura de vetor que armazena as amplitudes do espaço de estados. Ou seja, para sistemas com 24 *qubits*, 2 vetores (um para o estado atual e um para o próximo estado) contendo 2^{24} valores complexos são necessários.

6.2 Simulação Paralela

As validações e análise de desempenho iniciais da proposta de paralelização descrita neste trabalho é baseada na simulação de transformações *Hadamard*, as quais possuem maior custo computacional no ambiente *VPE-qGM*. Salienta-se que as simulações sempre consideram o estado global do sistema quântico, de forma que as otimizações sejam válidas quando da ocorrência de estados clássicos, em superposição ou emaranhados.

6.2.1 Metodologia

A comparação direta de desempenho neste trabalho se dá com relação aos resultados da simulação distribuída obtidos em (AVILA et al., 2012), os quais caracterizam o melhor desempenho para simulação de transformações *Hadamard* neste projeto até o momento. Comparações indiretas são posteriormente realizadas com o simulador *QuIDDPro*, uma vez que o esse simulador não está disponível abertamente para *download* e seu único referencial de desempenho está disponível em (GUTIERREZ et al., 2010).

A simulação distribuída foi realizada com dois *desktops* com ocupação de até 4 cores cada. Os nodos possuem a seguinte configuração: processador Intel Core2Quad, 4 GB RAM e rede Fast Ethernet. Uma máquina adicional foi utilizada como servidor para controle da simulação. A metodologia de testes para simulação distribuída considera a execução de 15 simulações de cada estudo de caso sobre as diferentes quantidades de cores utilizados nos nodos de processamento.

Para a simulação paralela com *GPU*, os componentes de *software* e *hardware* descritos na Tabela 5, foram utilizados. Transformações *Hadamard* de até 20 *qubits* foram simuladas. O principal parâmetro comparativo, obtido após 30 execuções pelo *NVIDIA Visual Profiler*, é o tempo de simulação para obtenção do novo estado do sistema.

Tabela 5: Configurações de hardware e software utilizadas

CPU	**GPU**	**Softwares**
Intel Core $i7-3770$	NVIDIA GT640 (GK 104)	Ubuntu 12.04 64 *bits*
8 GB RAM	1 GB DRAM	NVIDIA CUDA 5.0
	384 CUDA cores	NVIDIA Visual Profiler 5.0
	48 KB mem. compartilahda	PyCUDA 2012.1 (estável)

6.2.2 Análise de Resultados

A Tabela 6 contém os tempos de simulação para transformações *Hadamard* de até 17 *qubits*, considerando a simulação distribuída gerenciada pelo ambiente *VirD-GM* e a simulação paralela por *GPU* proposta neste trabalho.

Os resultados apresentados na Tabela 6 refletem a grande melhora no tempo de simulação quando da simulação por *GPUs*. Outras simulações de transformações *Hadamard* de até 20 *qubits* foram executadas apenas na *GPU*, gerando os seguintes tempos:

Tabela 6: Resumo da simulação distribuída utilizando o ambiente *VirD-GM* e a simulação paralela por *GPU*

	$H^{\otimes 15}$	$H^{\otimes 16}$	$H^{\otimes 17}$
1 core	138.338(s)	1126.529(s)	3075.729(s)
2 cores	72.824(s)	573.611(s)	1632.369(s)
4 cores	41.069(s)	304.008(s)	934.069(s)
8 cores	24.384(s)	160.049(s)	509.275(s)
Tempo GPU	0.210(s)	0.863(s)	3.326(s)

- $H^{\otimes 18}$: 13.070 segundos;
- $H^{\otimes 19}$: 52.071 segundos;
- $H^{\otimes 20}$: 210.548 segundos.

Esses testes não foram executados de forma distribuída devido ao elevado tempo de simulação necessário, porém as comparações já realizadas refletem a grande diferença no desempenho das duas abordagens.

Os *speedups* apresentados na Figura 37 atingiram um valor máximo de $\approx 1300\times$ quando comparados com a execução sequencial utilizando um *core* no ambiente *VirD-GM*. A abordagem baseada em *GPU* superou o melhor resultado para simulação distribuída no *VirD-GM* em $\approx 185\times$ para uma *Hadamard* de 16 *qubits*.

Esse desempenho se justifica pela quantidade de *CUDA cores* disponíveis na *GPU* utilizada neste trabalho, bem como pela arquitetura de memória que resulta em um elevado *throughput* de dados.

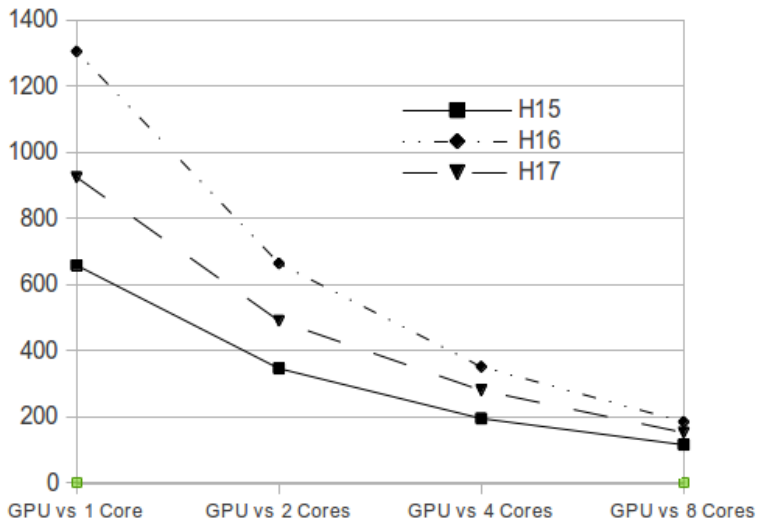


Figura 37: Speedups para a simulação por GPU em comparação com as diferentes quantidades de *CPU cores* na simulação distribuída gerenciada pelo *VirD-GM*.

6.3 Considerações Finais

A validação e análise de desempenho da simulação no ambiente *VPE-qGM* após todos os esforços relacionados às implementações descritas neste trabalho apresentam resultados que incentivam a contínua busca por novas possibilidades de otimização. Como característica marcante deste trabalho, destaca-se que embora a simulação quântica seja uma tarefa de alta complexidade computacional que usualmente exige recursos computacionais em larga escala, todos os resultados apresentados foram obtidos com o uso de apenas um computador com configuração modesta.

A possibilidade de descrição de transformações quânticos (controlados ou não) reduz o número de operações necessárias para realizar a evolução de estado do sistema quântico. Dessa forma, o incremento no desempenho viabilizou a simulação sequencial de algoritmos quânticos reversíveis (baseados em transformações quânticas controladas) com até 24 *qubits*. Em contrapartida, as simulações contendo apenas transformações *Hadamard*, a qual tem maior custo computacional no *VPE-qGM*, ficaram limitadas a sistema com até 14 *qubits* devido ao elevado tempo de simulação requerido.

Com o intuito de reduzir o tempo de simulação, neste estágio inicial apenas considerando transformações quânticas não controladas, a simulação paralela através de *GPUs* apresentou uma drástica redução no tempo de simulação. Devido ao grande número de *CUDA cores* utilizados, um *speedup* máximo na ordem de $\approx 1300\times$ foi obtido quando comparado com a simulação em um *core* no ambiente *VirD-GM*. Quando utilizados 8 - *cores* um *speedup* de $\approx 185\times$ foi obtido. Todos os estudos de caso considerados neste trabalho consideram transformações *Hadamard*.

Pela análise dos resultados gerados por este trabalho, conclui-se que mais um passo importante foi dado em direção a uma solução diferenciada para simulação de algoritmos quânticos. Considerando que várias novas otimizações ainda são aplicáveis, é possível perceber que novos resultados promissores ainda podem ser alcançados.

7 CONCLUSÃO

A computação quântica consiste em uma nova abordagem para realização de computações direcionadas a resolução de problemas não solucionáveis em tempo polinomial por qualquer estrutura computacional disponível atualmente. A complexidade inerente à interpretação dos fenômenos quânticos que proporcionam o ganho de desempenho prometido pela computação quântica, aliada ao custo dos cálculos para aplicação do produto tensorial, exigem o uso de modelos e ferramentas para auxiliar na modelagem e simulação de aplicações quânticas.

Os recentes avanços rumo a construção de *hardwares* quânticos justificam os esforços em direção à consolidação de plataformas para modelagem, simulação e interpretação de algoritmos quânticos, uma vez que, no estágio atual, computadores quânticos físicos estão nos princípios de desenvolvimento e ainda apresentam baixa escalabilidade, não suportando sistemas quânticos complexos. Dessa forma, o estudo e desenvolvimento de novos algoritmos pode ser auxiliado for modelos e simuladores quânticos, visando o contínuo desenvolvimento dessa frente de atuação.

A busca por uma nova alternativa dentre as soluções atualmente disponíveis se justifica por duas características encontradas e diferentes simuladores, porém ainda não integradas em um único ambiente. A primeira diz respeito às otimizações introduzidas pelo simulador *QuIDDPro*, as quais garantiram bons resultados no âmbito do consumo de memória porém para sistemas complexos o tempo de simulação ainda é alto, uma vez que a simulação é ocorre de forma sequencial. A segunda considera a abordagem de processamento paralelo utilizado por outras ferramentas (GUTIERREZ et al., 2010), (NIWA; MATSUMOTO; IMAI, 2008) e (RAEDT et al., 2006), na qual tem-se a possibilidade de redução no tempo de simulação porém a limitação se dá pelo consumo de memória, já que nesses casos otimizações mais para redução no uso de memória não são consideradas.

7.1 Principais Contribuições

Este trabalho busca estabelecer as diretrizes para consolidação dessa integração através de duas formas: modelando as transformações quânticas de forma a reduzir o consumo de memória pela geração dinâmica de elementos e, em um segundo momento, estabelecendo o correspondente suporte à execução paralela através de *GPUs*.

Tendo em visa um melhor aproveitamento dos recursos de memória RAM, a modelagem de transformações quânticas (controlados ou não) a partir dos construtores de Processos Quânticos e Processos Quânticos Parciais evita o aumento exponencial de armazenamento necessário para transformações multidimensionais. Além disso, esses construtores reduze o número de operações necessárias para realizar a evolução de estado do sistema quântico, uma vez que a padronização presente nas definições das transformações quânticas permitem que computações redundantes sejam evitadas. Dessa forma, viabilizou-se a **simulação sequencial** de algoritmos com até 24 *qubits*. Destaca-se que, esses resultados foram obtidos pela simulação de algoritmos quânticos reversíveis, os quais são baseados apenas em transformações controladas e, consequentemente, têm um custo de simulação menor no ambiente *VPE-qGM*.

Para simulação de algoritmos quânticos que exploram fenômenos de superposição e interferência quântica (GROVER, 1996) e (SHOR, 1997), os quais têm um grande custo computacional no ambiente *VPE-qGM* devido à aplicação de transformações *Hadamard* e várias partes do algoritmo, soluções interessantes podem ser obtidas pela exploração de recursos advindos da área de processamento paralelo. Neste trabalho, especificamente, tem-se a utilização de *GPUs* para realizar a paralelização dos cálculos decorrentes da execução de Processos Quânticos. Seguindo as premissas de otimização difundidas pela comunidade científica, uma grande melhora no desempenho da simulação foi obtida. Apesar de ainda não estar em um estágio ideal de desenvolvimento, os resultados da simulação paralela mostraram uma melhora de $\approx 185\times$ quando comparado com uma execução correspondente sobre 8 *cores* no ambiente *VirD-GM*.

O principal diferencial deste trabalho está na generalização da solução para simulação quântica apresentada. Por ser uma abordagem que sempre considera o estado global da simulação, independente de qual transformação quântica está sendo aplicada, a estratégia de execução das computações pode ser estendida para qualquer transformação quântica unitária. Os resultados obtidos são significantes para o projeto no qual está inserido, expandindo as capacidades de simulação do ambiente *VPE-qGM*. Os estudos realizados ainda são responsáveis por definir os próximos desafios a serem enfrentados na continuidade do trabalho.

7.2 Desafios em Aberto

As contribuições deste trabalho viabilizaram as primeiras execuções em *GPU* da biblioteca de simulação do *VPE-qGM*. Os desafios iniciais relacionados a estratégia de integração com as estruturas de simulação do ambiente foram superados. As primeiras simulações foram executadas com transformações quânticas não controladas considerando as otimizações mais importantes na área de *GPGPU*.

Entretanto, vários desafios seguem em aberto:

- Extensão das implementações para suporte às demais configurações de transformações quânticas;
- Otimização dos padrões de escrita do *CUDA kernel* para reduzir o número de transações de memória;
- Estudo de otimizações para o armazenamento dos dados associados ao vetor de estado do sistema quântico para menor consumo de memória.

REFERÊNCIAS

AARONSON, S. **Shtetl-Optimized**: Shor, I'll do It. http://scottaaronson.com/blog/?p=208.

AVILA, A.; MARON, A.; REISER, R.; PILLA, M. Estendendo o Ambiente VirD-GM para Execução Distribuída de Processos Quânticos. In: Anais do WSCAD 2012, 2012. p.1–4.

D. MASLOV, G. D.; SCOTT, N. **Reversible Logic Synthesis Benchmarks Page.** Disponível por WWW em <http://www.cs.uvic.ca/ dmaslov>(apr.2012).

FONSECA, V.; REISER, R.; YAMIN, A.; PILLA, P. VirD-GM: Towards a Grid Computing Environment. In: Proceedings of CCGRID 2007, 2007. p.1–6.

GIRARD, J.-Y. Between logic and quantic: a tract. In: LINEAR LOGIC IN COMPUTER SCIENCE, 2004. Cambridge University Press, 2004. p.466–471.

GROUP, Q. C. **QuIDDPro**: High-Performance Quantum Circuit Simulator. The University of Michigan, 2007. Disponível por WWW em <http://vlsicad.eecs.umich.edu/Quantum/qp/results.html> (dec.2011).

GROVER, L. A Fast Quantum Mechanical Algorithm for Database Search. **Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing**, p.212–219, 1996.

GUTIERREZ, E.; ROMERO, S.; TRENAS, M.; ZAPATA, E. Quantum Computer Simulation Using the CUDA Programming Model. **Computer Physics Communications**, p.283–300, 2010.

HENKEL, M. **Quantum Computer Simulation**: New World Record on JUGENE. HPC Wire, 2010. Disponível em <http://www.hpcwire.com/hpcwire/2010-06-28/quantum_computer_simulation_new_world_record_on_jugene.html> (fev 2013).

KLOECKNER, A. **PyCuda Online Documentation**. http://documen.tician.de/pycuda/.

KNILL, E. H.; NIELSEN, M. A. **Theory of quantum computation**. Kluwer, 2000. Comment: 5 pages.

MARON, A.; PINHEIRO, A.; REISER, R.; PILLA, M. Distributed Quantum Simulation on the VPE-qGM. In: Anais do WSCAD-SSC 2010. IEEE Computer Society - Conference Publising Services, 2010. p.128–135.

MARON, A.; PINHEIRO, A.; REISER, R.; PILLA, M. Consolidando uma Infraestrutura para Simulação Quântica Distribuída. In: Anais do ERAD 2011. SBC/Instituto de Informática UFRGS, 2011. p.213–216.

MARON, A.; PINHEIRO, A.; REISER, R.; YAMIN, A.; PILLA, M. Ambiente VPE-qGM: Em Direção a uma Nova Abordagem para Simulações Quânticas. **Revi. CCEI**, v.14, p.29–46, 2010.

MARON, A.; ÁVILA, A.; REISER, R.; PILLA, M. Introduzindo uma Nova Abordagem para Simulação Quântica com Baixa Complexidade Espacial. In: Anais do DINCON 2011. SBMAC, 2011. p.1–6.

NIELSEN, M. A.; CHUANG, I. L. **Computação Quântica e Informação Quântica**. Bookman, 2003.

NIWA, J.; MATSUMOTO, K.; IMAI, H. General-Purpose Parallel Simulator for Quantum Computing. **Lecture Notes in Computer Science**, v.2509, p.230–249, 2008.

NVIDIA. **Technololy Overview - NVIDIA GeForce GTX 680**. 2012. Version 1.0.

NVIDIA. **CUDA Programming Guide**. NVIDIA Corp., 2012. Version 4.2.

NVIDIA. **CUDA Community Showcase**. Disponível em <http://www.nvidia.com/object/cuda-apps-flash-new.html>.

PESSOA, O. **Conceitos de Física Quântica**. SP: Editora Livraria da Física, 2003.

PORTUGAL, R.; LAVOR, C.; MACULAN, N. **Uma Introdução à Computação Quântica**. SP: Notas em Matemática Aplicada - SBMAC, 2004.

RAEDT, K. D.; MICHIELSEN, K.; RAEDT, H. D.; TRIEU, B.; ARNOLD, G.; RICHTER, M.; LIPPERT, T.; WATANABE, H.; ITO, N. **Massive Parallel Quantum Computer Simulator**. Quantum Physics, 2006. http://arxiv.org/abs/quant-ph/0608239.

REISER, R.; AMARAL, R.; COSTA, A. Quantum Computing: Computation in Coherence Spaces. In: Proceedings of WECIQ 2007. UFCG - Universidade Federal de Campina Grande, 2007. p.1–10.

REISER, R.; AMARAL, R.; COSTA, A. Leading to Quantum Semantic Interpretations based on Coherence Spaces. In: Proceedings of NANOBIO 2007. Lab. de II - ICA/DDE - PUC-RJ, 2007. p.1–6.

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

THE Double-Slit Experiment. http://www.doubleslitexperiment.com/.

VIAMONTES, G. **Efficient Quantum Circuit Simulation**. 2007. Phd Thesis — The University of Michigan.