Simulação Distribuída de Algoritmos Quânticos via GPUs

Anderson B. de Avila[1], **Murilo F. Schumalfuss**[1],
Renata H. S. Reiser[1], **Mauricio L. Pilla**[1], **Adriano Kurz Maron**[2]

[1]Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas (UFPEL)
Caixa Postal 15.064 - 91.501-970 - Pelotas - RS - Brasil

{abdavila,mfschumalfuss,reiser,pilla}@inf.ufpel.edu.br

[2]Department of Computer Science
University of Pittsburgh
Sennott Square – Pittsburgh/PA, USA

akk48@pitt.edu

Abstract. *This work provides an extension of the D-GM environment to get distributed simulation of quantum algorithms via GPUs. The main contribution of this work consists in the optimization of the environment VirD-GM, conceived in two steps:* (i) *the theoretical studies and implementation of the abstractions of the Mixed Partial Process defined in the qGM model, focusing on the reduction of the memory consumption regarding multidimensional quantum transformations;* (ii) *and the distributed/parallel implementation of such abstractions allowing its execution on clusters of GPUs. The results obtained in this work embrace the distribute/parallel simulation of Hadamard gates up to* 21 *qubits and confirm the performance gain with the increase number of clients.*

Resumo. *Este trabalho visa a extensão do ambiente D-GM para simulação distribuída de algoritmos quânticos via GPUs. A principal contribuição deste trabalho é a otimização do ambiente VirD-GM, concebida em duas etapas:* (i) *o estudo teórico e implementação das abstrações de Processos Mistos Parcias definidos no modelo qGM, visando a redução no consumo de memória associado à transformações quânticas multidimensionais;* (ii) *e a implementação distribuída/paralela dessas abstrações para correspondente execução sobre clusters de GPUs. Os resultados obtidos neste trabalho contemplam a simulação distribuída/paralela de transformações Hadamard de até* 21 *qubits e comprovam o ganho de desempenho com o aumento do número de clientes.*

1. Introdução

A simulação de algoritmos quânticos em computadores clássicos viabiliza o desenvolvimento e teste de algoritmos quânticos, antecipando o conhecimento acerca de seu comportamento quando da execução sobre um *hardware* quântico. A simulação de sistemas quânticos através de computadores clássicos ainda se mostra um desafio de pesquisa em aberto, justificando o estudo de soluções voltadas para a simplificação no processo de modelagem e interpretação de algoritmos quânticos [Nielsen and Chuang 2000]. Mais significante, as otimizações no ganho de desempenho da simulação contribuem para o suporte a sistemas quânticos mais complexos.

O paradigma de programação *GPGPU* (*General Purpose Computing on Graphics Processing Units*) tornou-se recentemente uma das abordagens mais interessantes para *HPC* (*High Performance Computing*) devido ao seu bom equilíbrio entre custo e benefício. Henkel [Henkel 2010] explora *GPUs* para simulação quântica. Neste contexto, tem-se o espaço de memória disponível na *GPU* como principal restrição, uma vez que o tempo de simulação mostra-se bem reduzido. A inovação na área de pesquisa de computação/simulação quântica é integrar num mesmo ambiente essas duas abordagens: simulação distribuída e *GPU*. Nosso projeto visa consolidar o ambiente *D-GM* como suporte para esse cálculo e processamento híbrido da simulação quântica.

Neste sentido, a principal contribuição deste trabalho consiste no **aumento das capacidades de simulação do ambiente *VirD-GM* [Avila et al. 2014] pelo estudo e implementação de abstrações presentes no modelo *qGM* para interpretação de transformações quânticas a partir de Processos Mistos Parciais(*MPPs*) e a implementação distribuída/paralela dessas abstrações para correspondente execução sobre clusters de *GPUs***. Estas contribuições simbolizam os esforços iniciais para explorar a simulação de algoritmos quânticos usando arquiteturas híbridas.

Este artigo está estruturado da seguinte forma: a Seção 2 compreende os fundamentos da Computação Quântica (CQ). Na Seção 3 são descritos os principais trabalhos relacionados. A seção 4 abrange a distribuição das computações no VirD-GM. Na Seção 5 está descrito o funcionamento do ambiente *VirD-GM*, bem como as implementações realizadas. Em seguida, a Seção 6 descreve como se dá a execução paralela no ambiente. Por fim, na Seção 7 são apresentados os resultados obtidos das simulações distribuídas/ paralelas, seguido da Seção 8 com as principais conclusões obtidas a partir da realização deste trabalho e as propostas de continuidade.

2. Fundamentação

Na *CQ*, o *qubit* é a unidade básica de informação, definido por um vetor de estado, unitário e bidimensional, genericamente descrito, na notação de Dirac [Nielsen and Chuang 2000], pela expressão $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$.

Os coeficientes α e β são números complexos correspondentes às amplitudes dos respectivos estados, respeitando a condição de normalização $|\alpha|^2 + |\beta|^2 = 1$ e ainda, garantindo a unitariedade do vetor de estado do sistema, representado por $(\alpha, \beta)^t$. As amplitudes permitem que o sistema represente, simultaneamente, estados distintos, configurando um estado de **superposição quântica**, característica que origina o fenômeno do paralelismo quântico.

O espaço de estados de um sistema quântico de múltiplos *qubits* é compreendido pelo produto tensorial do espaço de estados de seus sistemas componentes. Considerando um sistema quântico de dois *qubits*, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ e $|\varphi\rangle = \gamma|0\rangle + \delta|1\rangle$, o espaço de estados é composto pelo produto tensor $|\psi\rangle \otimes |\varphi\rangle$, ou seja $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$.

A mudança de estado em um sistema quântico é feita por transformações quânticas (TQs) unitárias, associadas a matrizes quadradas ortonormalizadas de ordem 2^N, sendo N a quantidade de *qubits* da TQ.

3. Trabalhos Relacionados

Outros simuladores paralelos já foram propostos para acelerar a simulação de algoritmos quânticos. Se baseado em *clusters* ou *GPUs*, bons resultados podem ser obtidos. A seguir, são discutidos o estado-da-arte em simulação quântica de alto desempenho.

3.1. Simulação de Computação Quântica usando supercomputadores

O MPQCS (*Massive Parallel Quantum Computer Simulator*) descrito em [Raedt et al. 2006] é baseado em *MPI*, podendo ser aplicado a máquinas paralelas *high end* ou a aglomerados de *desktops* comuns. Os algoritmos são descritos através de um conjunto universal de Transformações Quânticas (TQs), por exemplo, $\{H, S, T, CNOT\}$. Através da combinação destas TQs, é possível descrever qualquer algoritmo quântico.

Ao utilizar qualquer conjunto universal de TQs, limitações nos processos de desenvolvimento podem surgir, já que operações mais complexas devem ser especificados apenas em termos dessas TQs.

No entanto, essa simplicidade permite a aplicação de otimizações mais agressivas no simulador, já que os padrões dos cálculos são mais previsíveis.

Sua realização mais recente aconteceu em 2010, conforme publicado em [Henkel 2010]. A simulação do algoritmo de *Shor* com 42 qubits foi realizada no supercomputador JUGENE, fatorando o número 15707 em 113×139. Para isso, foram necessários 262.144 processadores, porém o consumo de memória e o tempo de simulação requerido não foram divulgados.

3.2. Simulação de computação quântica usando o *framework CUDA*

O simulador quântico descrito em [Gutierrez et al. 2010] usa o *framework CUDA* para explorar a natureza paralela de algoritmos quânticos. Nesta abordagem, os cálculos relacionados com a evolução do sistema quântico são realizadas por milhares de *threads* dentro de uma *GPU*.

Esta abordagem considera um conjunto definido de TQs de um qubit e dois qubits, sendo uma solução mais geral em termos de operações ao comparar a proposta de [Raedt et al. 2006]. O uso de um conjunto mais expressivo de TQs expande as possibilidades para descrever os cálculos de um algoritmo quântico.

Dentre as principais limitações da simulação quântica com *GPUs*, destaca-se a capacidade de memória mais restritiva, limitando a simulação apresentada por [Gutierrez et al. 2010] a sistemas com um máximo de 26 qubits. Como uma importante motivação para essa abordagem, o tempo de simulação pode alcançar *speedups* de $95\times$ contra uma simulação muito otimizada em *CPU*.

3.3. Simulação Híbrida

Os recentes avanços na computação *GPGPU* resultaram em uma nova tendência para a computação paralela, que explora a cooperação entre CPUs e GPUs para executar com eficiência uma tarefa. Muitos algoritmos estão sendo adaptados para tirar proveito dessa nova arquitetura, mas nenhum simulador quântico foi apresentado até o momento.

Muitos desafios em relação à comunicação, programação e sincronização dos cálculos devem ser superados para produzir um simulador escalável sob esta abordagem híbrida. Apesar dos muitos desafios, a recompensa se justifica por duas razões principais:

(i) o poder computacional provido por clusters atualizado com co-processadores é muito maior do que *clusters* somente com *CPUs*;

(ii) a relação FLOP/Custo($) fazem das *GPUs* uma solução acessível para aumentar o poder computacional dos *clusters*.

Diante desse cenário, podemos afirmar que nossa proposta consiste numa solução inicial para explorar esta arquitetura computacional híbrida.

4. Granularidade e Distribuição da Computação

Nesta Seção são descritas as formas de distribuições providas pelo modelo *qGM*, que proporcionam a capacidade de definir diferentes *layouts* de computação para a mesma transformação quântica. Dentre estas, destaca-se neste trabalho a Distribuição Mista. Todas as representações do modelo D-GM consideram manter uma interpretação quântica coerente para simulação de transformações (totais/parciais) e estados (totais/parciais).

4.1. Distribuição de QPPs

Uma maneira de distribuir o fluxo de dados para cálculo de uma transformação é dividir o correspondente Processo Quântico(*QP*) em 2 ou mais Processos Quânticos Parciais(*QPPs*) [Avila et al. 2012] como na Figura 1.

QPPs permitem abstrair informações acerca das linhas da matriz da TQ. Assim, apenas algumas amplitudes do vetor de estado corrente são atualizados, aquelas que foram calculados a partir das linhas que compõem cada *QPP*. O valor $\perp$ nas linhas denota informações omitidas, de acordo com esta proposta de abstração de informação nos *QPPs*.

Nesta abordagem, o cálculo de cada *QPP* será realizado de forma independente para cada conjunto de amplitudes do estado corrente associado à TQ, sem necessidade de compartilhamento de dados para obter tais resultados. A quantidade de amplitudes atualizadas no vetor de estado corrente após execução de *QPP* caracteriza a granularidade desta computação.

A Figura 1 apresenta a modelagem dos *QPPs* associados ao operador Hadamard de segunda ordem, $H^{\otimes 2}$. A segunda coluna da Figura 1 ilustra um cálculo executado em 2 *QPPs*. Cada *QPP* calcula 2 amplitudes do vetor de estado. Nesta ilustração, a granularidade está representada pela tupla $(2, 4)$, ou seja, 2 linhas estão sendo atualizadas e 4 amplitudes do estado global estão sendo lidos. Assim, neste contexto, o mais elementar dos *QPPs* tem configuração de granularidade $(1, 4)$.

QPPs podem ser associados aos nodos em um sistema distribuído, mas é necessário que todos os nodos mantenham uma cópia atualizada do vetor representando o estado de entrada do sistema. Tal condição aumenta o consumo de memória nos nodos, sobrecarregando a rede durante a computação de uma TQ.

4.2. Distribuição de PCPs

A Distribuição de Processos Clássicos Parciais(*PCPs*) permite abstrair informações acerca das colunas da matriz da TQ. Assim, a partir dos *PCPs* são gerados resultados parciais para todas as amplitudes do vetor de estado resultante.

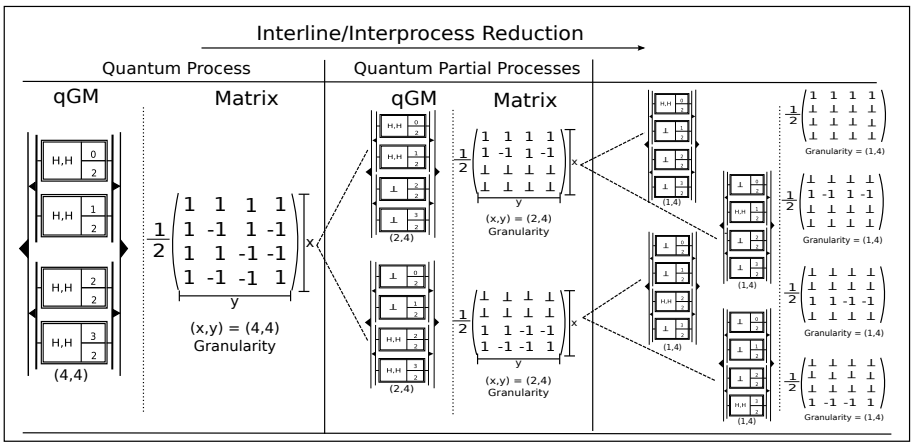

Figura 1. Distribuindo a computação do operador $H^{\otimes 2}$ em *QPPs*

Algumas distribuições em termos de *PCP* para o operador Hadamard $H^{\otimes 2}$ são apresentadas na Figura 2. O valor $\perp$ é gerado nas colunas que serão omitidas, de acordo com a proposta de abstração da informação referente a uma TQ.

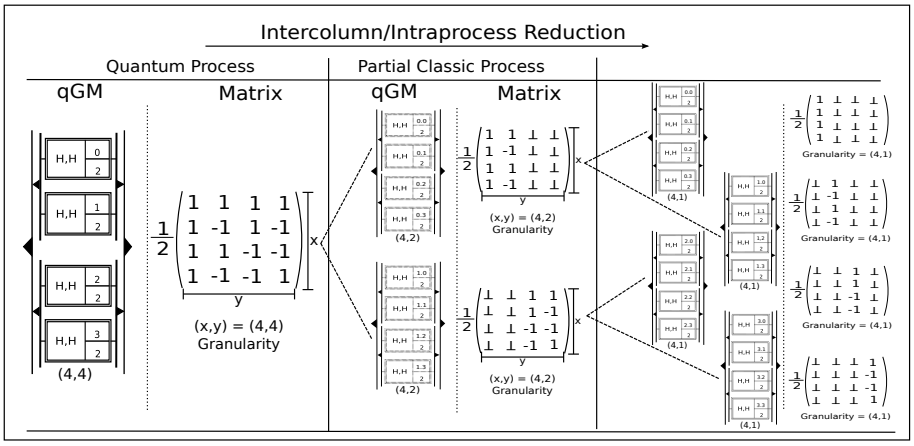

Figura 2. Distribuindo a computação da tranformação $H^{\otimes 2}$ em *PCPs*

De forma análoga ao texto descritivo da Figura 1, na Figura 2, a maior configuração de granularidade é dada por $(4, 1)$, sendo que cada sincronização irá gerar uma amplitude parcial para cada estado da base computacional. A computação de cada *PCP* é independente, podendo ser executada em paralelo, se houver memória suficiente para armazenar as amplitudes parciais.

Os *PCPs* também podem ser associados aos nodos em um sistema distribuído, e cada nodo recebe apenas uma porção do estado de entrada, inicializando o cálculo dos *PCPs* associados a TQ. Porém cada *PCP* gera um vetor de estado de saída, com valores parciais para o resultado do cálculo de cada amplitude, aumentando o consumo de memória nos nodos.

4.3. Distribuição Mista

A combinação da *Distribuição de QPPs e de PCPs* em um layout misto/integrado, contempla ambos desafios: o gerenciamento na sobrecarga da rede e a redução no consumo de memória. Nomeamos este layout de *MPP* (*Mixed Partial Process*).

Com a modelagem e implementação dos *MPPs* no VirD-GM, o programador tem maior controle do uso da memória e do gerenciamento dos cálculos (granulusidade das computações) considerando os recursos disponíveis. A integração deste gerenciamento com as interfaces do VPE-qGM, incrementam as simulações no ambiente D-GM.

Com as computações baseadas em *QPPs* e *PCPs*, tem-se um gerenciamento do limite nas amplitudes acessadas e nos resultados destas computações. Assim, o controle de granulosidade da computação de cada sincronização pode ser feita de acordo com a memória disponível no nodo de processamento.

Na Figura 3, ilustram-se exemplos de layouts de computação obtidos pela distribuição mista/integrada, considerando um operador Hadamard $H^{\otimes 2}$. Neste contexto, pela aplicação da redução mista no processamento da matriz associada, tem-se a redução da granulosidade de $(4, 4)$ para $(2, 2)$ na distribuição da correspondente computação envolvendo *QPPs* e *PCPs*.

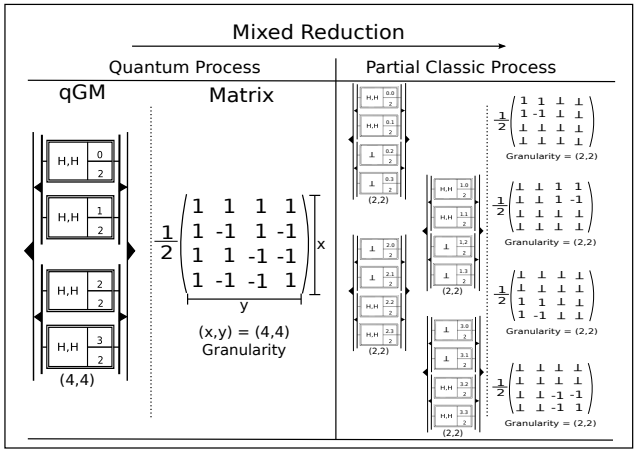

Figura 3. Distribuição mista da computação do operador $H^{\otimes 2}$

A Figura 4 ilustra como tal distribuição seria realizada. O **Nodo 1** executa um cálculo sobre apenas metade das amplitudes do vetor de estado, gerando assim amplitudes parciais para as correspondentes posições do vetor de estado. O **Nodo 2** executa o cálculo complementar do **Nodo 1**. No final, as amplitudes parciais de cada estado são enviados ao servidor para se obter o estado final do sistema quântico, o qual é obtido pela aplicação de operadores (composição, somatório e/ou outros).

5. VirD-GM e Implementações

O ambiente *VirD-GM* se encontra em desenvolvimento com o objetivo de realizar um gerenciamento transparente da execução distribuída no ambiente *VPE-qGM* [Maron et al. 2013]. Nesta seção é descrito o funcionamento do ambiente *VirD-GM* para suporte a modelagem da computação mista e as implementações dos *MPPs*.

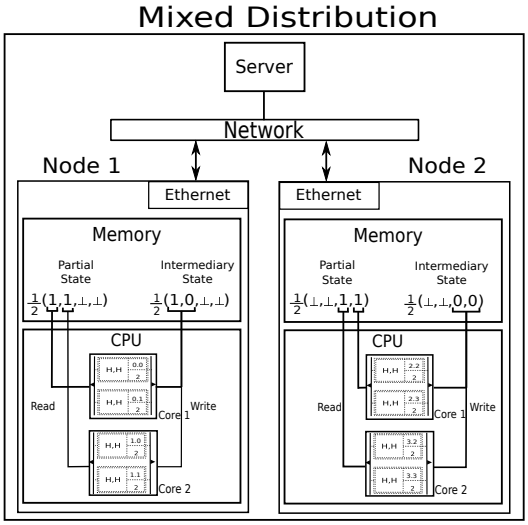

Figura 4. Configuração para uma distribuição mista em um *cluster*

A execução distribuída de aplicações a partir do *VirD-GM* exige, como parâmetros de entrada, um arquivo descritor de processos, um arquivo descritor de memória e os endereços dos nodos de execução. Os principais módulos do ambiente *VirD-GM* são:

(i) *VirD-Loader*: é responsável pela interpretação de arquivos descritores contendo o algoritmo a ser simulado e seu vetor de estado inicial.

(ii) *VirD-Launcher*: realiza o escalonamento e controle de fluxo de execução das tarefas.

(iii) *VirD-Exec* controla a comunicação e transferência de dados entre os clientes de execução, nomeados *Vird-Clients*.

A simulação distribuída de um algoritmo quântico a partir de $MPPs$ consiste no envio de cada MPP para um *VirD-Client* pela aplicação do método *send*. Ao passar os parâmetros do MPP e a porção necessária da memória atual, o *VirD-Client* realiza a computação e gera uma lista com os valores parciais calculados, a qual é enviada para o *VirdServer* através do método *updateMemory*. E assim, o *VirdServer* atualiza a sua memória com os dados contidos nesta lista.

Este processo é feito para todos os arquivos temporários gerados na simulação, sendo necessário um arquivo para cada passo da simulação. Após todos os passos terem sido executados, o método *exportResult* é chamado, o qual é responsável por salvar o resultado obtido no arquivo *tempMemory.xml*. Este resultado que será posteriormente utilizado pelo *VPE-qGM* para carregar o resultado da simulação, o qual será apresentado nas interfaces dos seus editores. No Projeto D-GM, o módulo *VirD-GMConnection* gera os arquivos descritores e configura/executa os comandos de disparo de aplicações.

5.1. Implementação

Esta implementação visa contribuir para redução no aumento exponencial do consumo de memória gerado pelo produto tensor em simulações multi-qubits.

A estratégia desta implementação distribuída se baseia em que um *MPP* pode ser definido a partir da construção de matrizes básicas de menor ordem, as quais podem

ser combinadas de forma a gerar dinamicamente os elementos correspondentes à matriz resultante do produto tensorial. As linhas e colunas da matriz de uma TQ modeladas por *MPPs* definem quais amplitudes do vetor de estado são necessárias para realizar a computação e também quais estados parciais serão gerados para execução distribuída.

MPPs complementares (que interpretam conjuntos disjuntos de linhas e colunas da mesma matriz associada a uma TQ) podem ser agrupados no arquivo descritor de processos a fim de computar uma TQ.

As principais alterações que viabilizaram a extensão do ambiente VirD-GM para fornecer suporte a *MPPs* estão brevemente descritos logo a seguir.

- (i) Extensão do módulo *VirD-Loader*, visando carregar a memória de simulação em formato compatível para comunicação com a *GPU*, uma vez que Java não possui o tipo primitivo *complex*.
- (ii) Reestruturação do arquivo descritor de processos, modificado para interpretação e reconhecimento de um *MPP*.
- (iii) Reestruturação de estruturas de dados, transformado estruturas do tipo *complex* em estruturas do tipo *Float*, porque quando dois números *Floats* consecutivos são enviados para o kernel é possível interpretá-los como um único número complexo.
- (iv) Reestruturação do módulo *Vird-Exec*, enviando para um *VirD-Client* somente com a porção da memória que este *MPP* necessita para a simulação.
- (v) Extensão da biblioteca *QGM-Analyzer*, responsável pela maior parte da computação, viabilizando invocar o kernel *CUDA* quando solicitado pelo gerenciamento a execução da computação na *GPU*. Utiliza-se o *framework JCuda* [Yonghong et al. 2009] para invocação do kernel e comunicação com a *GPU* através de métodos.

6. Execução Paralela de MPPs

Este seção introduz os esforços que viabilizam a execução distribuída/paralela de *MPPs* utilizando o ambiente *VirD-GM*.

6.1. Estruturas de Dados

Os *MPPs* podem ser definidos 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. Na execução de um *MPP*, a biblioteca de execução faz uso dos seguintes parâmetros:

- (i) Lista de Matrizes (*matrices*): matrizes básicas geradas pelo *host-code*, não armazenando valores nulos;
- (ii) 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;
- (iii) 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;
- (iv) Última matriz de posições (*lastPositions*): armazena estrutura exclusiva para simplificar a indexação de elementos constantemente acessados.

(v) Início das linhas: (*initLines*): armazena em que posição da lista de matrizes se encontra o primeiro elemento das linhas das matrizes.
(vi) Tamanho das linhas (*widthLines*): armazena o quantidade de elementos em cada linha das matrizez;
(vii) Dimensão das matrizes (*dimensions*): armazena a dimensão de cada matriz.
($viii$) Lista de qubits (*exp*): armazena a lista de qubits associados a cada matriz.

Todos os valores correspondentes às matrizes de uma TQ são armazenados na forma de 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 6.3.

6.2. 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 se torna uma opção adequada. As cópias de dados são realizadas a partir de chamadas realizadas pelo framework *JCuda*.

Além dos dados referentes a parametrização dos *MPPs*, dois vetores de estado são encontrados no espaço de memória do *host*: (i) um modelando o estado atual do sistema quântico; e (ii) outro modelando o próximo estado, o qual será calculado. Antes da execução do *CUDA kernel* para início da computação, ambos vetores são copiados para a memória global da *GPU*.

6.3. CUDA Kernel

O *CUDA kernel* é implementado de forma iterativa, 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* mantém informações interna e controles de execução para definir os limites de acesso em cada matriz.

As informações constantes, que são comuns a todas às *CUDA threads* da aplicação, são definidos em tempo de execução alterando o arquivo base do *kernel*, onde estes valores ainda não estão definidos pois estas informações variam de execução para execução de acordo com o *MPP* à ser calculado. Na sequência, realiza-se a compilação do *kernel* resultante. Estas informações são: $BITS$, equivalente à 2 elevado ao número de qubits do *MPP*, é utilizado em operações de cálculos da posição do elemento nas matrizes; AMP informa quantas amplitudes cada *CUDA thread* calcula; $SHIFT\_READ$ e $SHIFT\_WRITE$ definem a posição de leitura e escrita base, repectivamente, do *MPP*; TAM_BLOCK define o tamanho do bloco; $TOTAL\_ELEMENTS$, $LAST\_MATRIX\_ELEMENTS$, $STACK\_SIZE$ e $STACK\_LINES$ são os parâmetros dos vetores da memória contante.

A computação de cada *thread* pode ser dividida em cinco passos:

Passo 1: **Inicialização** do vetor das novas amplitudes e das variáveis iniciais de cada thread, onde o único valor que diverge entre elas é o $lineId$, o qual define qual linha de cada matriz a thread irá acessar.

```
ind = d = dim = readValue = columnId = 0;
cuFloatComplex value = make_cuFloatComplex(1, 0);
for (i = 0; i < AMP; i++)
  newAmplitudes[i] = make_cuFloatComplex(0, 0);
lineId = (TAM_BLOCK * blockIdx.x + threadIdx.x) *
         dimensionC[STACK_SIZE] + SHIFT_WRITE;
```

Passo 2: **Avanço nas matrizes**, análoga a passo recursivo provendo multiplicações parciais entre os elementos indexados e valores calculados usando *lineId* e *columnId*.

```
while (ind >= 0){
  for ( ; ind < (STACK_SIZE); ind++){
    mask = dimensionC[ind] - 1;
    d += expC[ind];
    l = (lineId >> (BITS - d)) & mask;
    c = (columnId >> (BITS - d)) & mask;
    dim += dimensionC[ind];
    count = initLinesC[dim - dimensionC[ind] + l] + c;
    readValue = (readValue << (expC[ind])) | (positionsC[count] & mask);
    value = cuCmulf(value, matricesC[count]); }
  ind--;
```

Passo 3: **Atualização parcial das novas amplitudes do sistema** ao percorrer a última matriz a partir dos valores parciais calculados.

```
mask = dimensionC[STACK_SIZE] - 1;
readValue = readValue << expC[STACK_SIZE];
for( i = 0; i < LAST_MATRIX_ELEMENTS; i++){
  count = lastPositionsC[i] >> expC[STACK_SIZE];
  newAmplitudes[count] = cuCaddf(newAmplitudes[count], cuCmulf(cuCmulf(value, lastMatrixC[i]),
                         readMemory[(readValue | (lastPositionsC[i] & mask)) - SHIFT_READ])); }
readValue = readValue >> expC[STACK_SIZE];
```

Passo 4: **Atualização dos índices das colunas** 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.

```
teste = 1;
while ((ind >= 0) && teste){
  mask = dimensionC[ind] - 1;
  c = (columnId >> (BITS - d)) & (mask);
  l = (lineId >> (BITS - d)) & (mask);
  count = initLinesC[dim - dimensionC[ind] + l] + c;
  value = cuCdivf(value, matricesC[count]);
  readValue = readValue >> expC[ind];
  dim -= dimensionC[ind];
  if ((c + 1) < widthLinesC[dim + l]){
    teste = 0;    columnId += (1 << (BITS - d));    d -= expC[ind]; }
  else{    columnId -= (c << (BITS - d));    d -= expC[ind];    ind--; }}
```

Passo 5: **Cópia de dados**, com as novas amplitudes calculadas para a vetor de estados de escrita na memória global da *GPU*.

```
lineId = (TAM_BLOCK * blockIdx.x + threadIdx.x) * dimensionC[STACK_SIZE];
for (i = 0; i < AMP; i++)
  writeMemory[lineId + i] = newAmplitudes[i];
```

Os passos descritos compõem o *CUDA kernel* e definem a computação de uma *thread*, recalculando n amplitudes do novo vetor de estado de acordo com as TQs aplicadas, onde n é equivalente a dimensão da última matriz.

7. Resultados

Para validação e análise de desempenho da simulação de algoritmos quânticos a partir de $MPPs$ no *VirD-GM*, foram considerados estudos de casos com TQs *Hadamard* de até 21 *qubits* ($H^{\otimes 18}$, $H^{\otimes 19}$, $H^{\otimes 20}$ e $H^{\otimes 21}$), foi escolhido o operador *Hadamard* pois corresponde a uma TQ (com densa matriz associada) que apresenta maior carga de execução no ambiente de simulação. A metodologia adotada para a simulação distribuída via *GPU* considera a execução de 10 simulações de cada instância do operador *Hadamard*, considerando cada uma das configurações dentro do *cluster* de *GPUs*, sendo que o número de *MPPs* nunca ultrapassa o número de *GPUs*.

Os testes foram realizados em quatro desktops, sendo dois desktops com processador Intel Core i7, 8 GB de RAM e uma GPU NVIDIA GT640 e dois desktops com processador Intel Core i7, 8 GB de RAM e uma GPU NVIDIA GTX560. Uma máquina adicional, conectada ao cluster por uma rede *Fast Ethernet*, foi necessário para executar o servidor do *Vird-GM*. Como componentes de software, tem-se: JCUDA 0.5.0a, NVIDIA CUDA TOOLKIT 5.0, e Ubuntu 12.04 64 bits. Para as simulações com 1 e 2 clientes, os dois desktops utilizados foram os com GPU NVIDIA GT640.

As configurações de *MPPs* usadas são descritas na forma $R - W$, onde R é o número de partes em que a memória de leitura foi particionada e W é o número de partes em que a memória de escrita foi particionada. E o número de clientes usados para cada configuração é equivalente a $R \times W$, que é o número de *MPPs* necessários para a simulação completa da TQ naquela configuração, possuindo todas as combinações entre as memórias de leitura e escrita particionadas. O desvio padrão máximo de $0,81\%$ foi medido para a $H^{\otimes 19}$ com configuração $1 - 4$.

Na Figura 5, tem-se os speedups obtidos para os diferente tipos de configurações, com relação a configuração $1 - 1$, que usa um cliente. Salienta-se o ganho de desempenho com o aumento do número de clientes, independente da configuração utilizada, e o *speedup* se aproxima do ideal com o aumento do número de qubits das TQs. Isto ocorre porque o tempo gasto pela comunicação entre servidor e clientes se torna menos significativo com relação ao tempo gasto em execução no cliente.

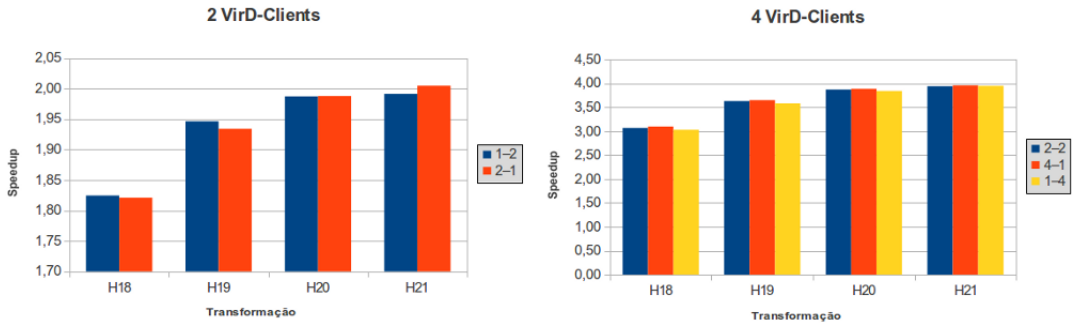

Figura 5. Speedup relativo à configuração $1 - 1$

8. Conclusão

A principal contribuição proposta pela extensão do VirD-GM é prover uma solução para o problema da alta complexidade espacial e temporal da simulação de algoritmos

quânticos. A modelagem de tranformações quânticas através de *MPPs* possibilitou agrupar computações parciais que compartilham um mesmo subconjunto de amplitudes, assim ao atribuir tal sincronização para um nó de processamento, evita-se o acesso a todo o vetor de estado, reduzindo a complexidade espacial associada. O desenvolvimento da extensão para suporte a simulação quântica distribuída a partir de *GPUs* utilizando o ambiente *VirD-GM* constitui uma solução para redução do tempo de simulação.

Os resultados obtidos mostraram que é possivel obter ganho de desempenho com o aumento do número de clientes e que tranformações com diferentes configurações, mas que utilizam o mesmo número de clientes, possuem um tempo de simulação semelhante, isto implica que o programador possui liberdade para definir a configuração dos *MPPs* que lhe parece melhor de acordo com os recursos disponíveis, sem se preocupar em ter uma grande perda de desempenho.

Trabalhos futuros em nosso projeto estão descritos nos seguintes tópicos: (i) suporte para portas controladas, projeções e operações de medida na abordagem distribuída; (ii) concepção e implementação do cliente de execução híbrido, em que o cálculo será executado por *CPUs* e *GPUs* de forma distribuída.

Referências

Avila, A., Maron, A., Reiser, R., and Pilla, M. (2012). Extending the VirD-GM environment for the distributed execution of quantum processes. In *Proceedings of the XIII WSCAD-WIC*, pages 1–4.

Avila, A., Maron, A., Reiser, R., and Pilla, M. (2014). Gpu-aware distributed quantum simulation. In *Proceedings of 29th Symposium On Applied Computing*, pages 1–6.

Gutierrez, E., Romero, S., Trenas, M., and Zapata, E. (2010). Quantum computer simulation using the cuda programming model. *Computer Physics Communications*, pages 283–300.

Henkel, M. (2010). Quantum computer simulation: New world record on jugene. Available at http://www.hpcwire.com/hpcwire/2010-06-28/ quantum _ computer_simulation_new_world_record_on_jugene.html (feb. 2013).

Maron, A., Reiser, R., and Pilla, M. (2013). High-performance quantum computing simulation for the quantum geometric machine model. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 474–481.

Nielsen, M. A. and Chuang, I. L. (2000). *Quantum Computation and Quantum Information*. Cambridge University Press.

Raedt, K. D., Michielsen, K., Raedt, H. D., Trieu, B., Arnold, G., Richter, M., Lippert, T., Watanabe, H., and Ito, N. (2006). Massive parallel quantum computer simulator. http://arxiv.org/abs/quant-ph/0608239.

Yonghong, Y., Grossman, M., and Sarkar, V. (2009). Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par 2009*, pages 1–13, Delft,Netherlands.