Optimizing Quantum Simulation for Heterogeneous Computing: a Hadamard Transformation Study

Anderson B. de Avila¹, Murilo F. Schumalfuss¹, Renata H. S. Reiser¹, Mauricio L. Pilla¹, Adriano K. Maron²

¹ Centre for Technological Development – Federal University of Pelotas
Pelotas/RS - Brazil

² Department of Computer Science
University of Pittsburgh Sennott Square – Pittsburgh/PA, USA
E-mail: {abdavila, mfschumalfuss, reiser, pilla}@inf.ufpel.edu.br, akk48@pitt.edu

Abstract. The D-GM execution environment improves distributed simulation of quantum algorithms in heterogeneous computing environments comprising both multi-core CPUs and GPUs. The main contribution of this work consists in the optimization of the environment VirD-GM, conceived in three 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 QTs; (ii) the distributed/parallel implementation of such abstractions allowing its execution on clusters of GPUs; (iii) and optimizations that predict multiplications by zero-value of the quantum states/transformations, implying reduction in the number of computations. The results obtained in this work embrace the distribute/parallel simulation of Hadamard gates up to 21 qubits, showing scalability with the increase in the number of computing nodes.

1. Introduction

Although quantum computers are restricted to specific research centers and laboratories, the results from quantum information and quantum computation (QC) are a reality nowadays, and the first steps towards quantum hardware have been an important research subject [3, 4].

In many aspects, the foundations of QC are still seen as inaccessible or difficult as well as an instigating subject by congregating areas such as Physics, Mathematics, Electrical and Electronics Engineering. Additionally, QC includes relevant research in Computer Science and Computer Engineering by providing new concepts which can be improved with the study of quantum algorithms.

The investigation of properties such as superposition and entanglement leads to the development of new mathematical models and, consequently, quantum algorithms may be designed through the mathematical description of the system and by using simulation software. Moreover, its ideas may be applied to stimulate thinking about new mathematical tools and to solve open problems in conventional Computer Science.

The simulation of quantum algorithms in classic computers allows for development and testing that would otherwise be impossible without proper quantum hardware. However, there are many aspects of simulating quantum computers that are still open research topics, mainly due...
to the high temporal and spatial complexity of emulating quantum phenomena without proper optimizations, which restrict unoptimized simulation to a few quantum bits, or qubits.

The explosion of states and operations required to simulate quantum algorithms with even a small number of qubits seem to be a good match for manycore architectures. The current available architectures such as general-purpose computing on graphics processing units (GPUs) present a high number of processing elements well-suited for the regular operations found in quantum computing simulation, but the memory hierarchy in current designs is limited and is not usually transparent, i.e. the programmer must manage allocation of variables to different kinds of memory to achieve performance.

Further improvements in the complexity of simulated algorithms can be severely hampered by memory constraints, and thus adding distributed resources to the simulation becomes a necessity.

1.1. Research Problem
Composition and synchronization of quantum operators enable the exploration of parallel and distributed computations to model the quantum parallelism. However, the spatial and temporal complexity of simulating quantum computing increases exponentially with the number of qubits and operations. Due to such behavior, implementations based only on vector and matrix notations are not practical for even a small numbers of qubits.

In our project, we intend to integrate distributed and GPU-based simulation in a single environment, the Distributed Geometric Machine (D-GM).

In previous works, optimizations for an efficient representation of multidimensional Quantum Transformations (QTs) were investigated in the D-GM environment [1, 15, 10, 11]. They contemplated the specification and implementation of D-GM environment and its main components, VPE-qGM and VirD-GM, enabling the modeling and distributed simulation of quantum algorithms, showing the evolution of quantum systems from a set of graphical interfaces.

1.2. Main Contribution
The general proposal to be achieved with the D-GM environment is the development of new strategies for simulation of quantum algorithms, focusing on the reduction of the spatial and temporal complexity by considering partial processes in the definition of quantum transformations.

In addition, based on the partiality concept, the granularity of distributed/parallel computation is investigated by considering classes of partial processes:

(i) the Classical Partial Processes (CPPs), related to partitions of read memory, which are represented by columns of its matrix representation;

(ii) the Quantum Partial Processes (QPPs), considering the write memory in the global memory state, represented by rows of the corresponding matrix representation; and

(iii) the Mixed Partial Processes (MPPs), obtained by integration of the two concepts above.

In this sense, the results contribute to integrate an efficient scheduling with more opportunity for performance gains, and new methods to balance communication and synchronization.

In particular, this work aims to extend the possibilities of environmental simulation capabilities of the D-GM, introducing optimizations for representation and simulation of parallel/distributed quantum transformations conceived in three steps:

(i) the theoretical studies and modeling of the abstractions of the MPPs, focusing on the reduction of the memory footprint for multidimensional quantum transformations;
(ii) the distributed/parallel implementation of such abstractions, allowing their execution on clusters of heterogeneous (non-uniform) computers with GPUs; and

(iii) optimizations that predict multiplications by zero-value amplitudes of the quantum state of quantum transformations which avoid unnecessary operations were developed, implying reduction in the number of computations.

The results achieved with the development of this work contribute for the consolidation of a more general objective – quantum simulations performing on hybrid computational system in the presence of hardware heterogeneity.

1.3. Paper Organization

This paper is structured as follows: Section 2 comprises the basics of Quantum Computing (QC). Section 3 describes the state-of-the-art in QC simulation. Afterwards, Section 4 presents an overview of the D-GM environment showing how computation is divided in the VirD-GM environment. Section 5 presents the modeling of modifications of VirD-GM environment and corresponding optimizations implemented in the current work. Section 6 details the parallel execution in the VirD-GM environment. Results are discussed in Section 7. Finally, conclusions and future work are drawn in Section 8.

2. Foundations of Quantum Computing

In QC, the qubit is the basic unit of information, defined by an unitary and bi-dimensional state vector that can be described in the Dirac’s notation [12] by

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle. \quad (1)$$

The coefficients $\alpha$ and $\beta$ are complex numbers corresponding to the amplitudes of the respective states, respecting the normalization condition $|\alpha|^2 + |\beta|^2 = 1$ and guaranteeing the unity of the system state vector represented by $(\alpha, \beta)^T$. The amplitudes allow the representation of distinct states at the same time, configuring quantum superposition states that originate the quantum parallelism.

The state space of a quantum system with multiple qubits is obtained by the tensor product of the space states of its subsystems. Considering a quantum system with two qubits, $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|\varphi\rangle = \gamma|0\rangle + \delta|1\rangle$, the state space comprehends the tensor product $|\psi\rangle \otimes |\varphi\rangle$, described by

$$|\psi\rangle \otimes |\varphi\rangle = \alpha \cdot \gamma|00\rangle + \alpha \cdot \delta|01\rangle + \beta \cdot \gamma|10\rangle + \beta \cdot \delta|11\rangle. \quad (2)$$

The state transition in an $N$-dimensional quantum system is performed by quantum transformations (QTs) defined by square matrices of order $N$ (i.e., $2^N$ components since $N$ is the number of qubits in the system). As an example, the matrix notation for the unitary QTs identity, Hadamard, Phase Gate, T Gate and Pauli X transformations [12, 8] are, respectively, defined by

$$Id = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & j \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 \\ 0 & \exp^{j\pi/4} \end{pmatrix} \quad \text{and} \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$  

The application of such transformations to a quantum state $|\psi\rangle$ generates the new corresponding global states $H|\psi\rangle = \frac{1}{\sqrt{2}} (\alpha + \beta)|0\rangle + \frac{1}{\sqrt{2}} (\alpha - \beta)|1\rangle$ and $X|\psi\rangle = \beta|0\rangle + \alpha|1\rangle$.

Additionally, QTs simultaneously applied to different qubits imply in the tensor product of the corresponding matrices, as described for the Hadamard transformation in Eq. (3):

$$H^{\otimes 2} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} H & H \\ H & -H \end{pmatrix}. \quad (3)$$
Besides the unitary transformations, controlled transformations are also preserved by the tensor product, modifying the state of one or more qubits considering the current state of other qubits in a multidimensional quantum state [14].

The CNOT quantum transformation receives the tensor product of two qubits $|\psi\rangle$ and $|\phi\rangle$ as input and applies the NOT (Pauly X) transformation to one of them (target qubit), considering the current state of the other (control):

$$CNOT = \begin{pmatrix} I_d & 0 \\ 0 & X \end{pmatrix}.$$  \hspace{1cm} (4)

The CNOT controlled transformation applied to a generic two-dimensional superposition quantum state $|\psi\rangle \otimes |\phi\rangle$ results in the following quantum state:

$$CNOT(|\psi\rangle \otimes |\phi\rangle) = \alpha \cdot \gamma |00\rangle + \alpha \cdot \delta |01\rangle + \beta \cdot \delta |10\rangle + \beta \cdot \gamma |11\rangle.$$  \hspace{1cm} (5)

Controlled transformations can be generalized in a similar way, e.g. the Tofolli and Fredkin transformations [12, 7].

3. Related Work

There is a number of parallel simulators trying to tackle the complex issues with efficient quantum computing simulator. Most approaches use clusters or GPUs to achieve reasonable performance and system complexity with different results. We discuss the most relevant works in the next subsections.

3.1. Quantum Simulation in Supercomputers

The Massive Parallel Quantum Computer Simulator (MPQCS) [13] uses the MPI library for communication, facilitating its execution on many parallel machines with or without shared memory. Quantum algorithms are described from a universal set of QTs such as $\{H, S, T, CNOT\}$. Combinations of these transformations may describe any quantum algorithm. This design choice promotes optimizations in the simulator, as the required calculations are predictable. However, designing and implementing quantum algorithms with higher complexity is bound to suffer from the lack of higher abstractions.

The most recent publication in a similar environment dates from 2010 [6], where the simulation of the Shor algorithm with 42 qubits was executed on the JUGENE supercomputer and the number 15707 was factored as $113 \times 139$. This simulation required 262,144 processors, without specifying memory usage and execution time.

3.2. GPU Quantum Simulation

Gutierrez et al. [5] used the CUDA framework to explore the parallel nature of quantum algorithms. In their approach, calculations required for the evolution of the quantum system are executed by thousands of threads in a GPU. A set of QTs of one and two qubits are more expressive than those found in Raedt et al. [13], however, the bigger the set of QTs, the more complex is the simulation algorithm due to the variety of patterns that are possible in the definition of a multi qubit transformation.

The main limitation with GPU approaches is related to the memory capacity, which are usually restricted to a few gigabytes. Gutierrez et al. [5] were able to simulate systems with 26 qubits at most with the 768 MB available in the GeForce 8800 GTX. When compared to extremely optimized simulations in general-purpose CPUs, speedups of 95 times are possible.
3.3. Hybrid Quantum Simulation

Advances in the programming environments and capabilities of both GPUs and CPUs resulted in a new tendency towards exploration of heterogeneous resources to further improve efficiency. Therefore, many applications are being adapted to take advantage of these new architectures. As far as the authors know, ours is the first integrated and hybrid environment for simulation of quantum computing.

Although the advantages of such hybrid environments are clear, there are many challenges related to communication, programming, and synchronization that must be overcome to make it feasible. There are two main justifications for facing these challenges:

(i) the computational power found in clusters with GPUs is much greater than those with only CPUs;

(ii) the cost/benefit (both in terms of performance and energy consumption) for applications well-suited for GPUs is much better than for CPUs.

In previous works, we presented how our approach can be used for computing of the Hadamard Transformation \([9, 2]\) in a GPU environment. In the current work, we extend our previous work to improve efficiency and scalability to the quantum applications performed up to now on heterogeneous platforms and further on hybrid platforms.

4. GPU-Aware Distributed Quantum Simulation

The Distributed Geometric Machine (D-GM) project proposes a complete framework for simulation of quantum algorithms. Different graphical interfaces provide tools for modeling the application and simulating it, either sequentially or in parallel, using GPUs or multicore CPUs.

Figure 1 illustrates how the D-GM Simulation Framework is organized. Each different level has its own purpose towards the unification of the modeling/simulation process. In the Quantum Circuit Level, one can describe the application following the quantum circuit model and then automatically export it to a corresponding representation in the qGM (Quantum Geometric Machine) model \([11]\). The qGM level contains the VPE-qGM (Visual Programming Environment for the qGM Model) environment that allows the user to describe and simulate the computations under the qGM model. In the D-GM level, the distributed simulation manager, VirD-GM (Virtual Distributed Geometric Machine) – seamlessly handles tasks such as communication, scheduling, and synchronization when a distributed simulation is required. Finally, the hardware level enlists all the devices that can be explored by this framework, from regular desktops for sequential simulation to clusters with multiple GPUs.

The VirD-GM environment \([9, 2]\) provides transparent management of the distributed execution for the VPE-qGM environment.

Distributed execution of applications from VirD-GM requires the following inputs: (a) a process description file, (b) a memory description file, and (c) the addresses of the execution nodes.

The main modules found in the VirD-GM environment are:

(i) VirD-Loader: responsible for the interpretation of the input files;

(ii) VirD-Launcher: executes scheduling and synchronization of tasks; and

(iii) VirD-Executor: controls communication and data transfer among execution nodes (VirD-Clients).

Currently, there are three distribution options provided by the D-GM environment that allow for the definition of different computing layouts for the same quantum transformation which maintain a coherent quantum interpretation for the simulation of total/partial transformations,
as well as total and partial states. Each computing layout defines a method to divide and execute the operations that are required for a complete quantum computing simulation. They are described in the next subsections.

4.1. Distribution of Quantum Partial Processes

A Quantum Partial Process (QPP) [1] abstracts information about the rows of a matrix for a quantum transformation. Hence, only a few amplitudes from the current state vector are updated – those that were calculated from the rows that compose each QPP. The \( \perp \) value in the rows denotes the omitted information. In this approach, the calculation of each QPP will be executed in an independent way for each set of amplitudes from the current state associated to the quantum transformation, without requiring shared data. The number of amplitudes updated in the current state vector after the execution of each QPP characterizes the granularity of the computation.

Figure 2 shows the division of a Quantum Process (QP) defining a Hadamard transformation of second order, \( H^{\otimes 2} \), in QPPs. The first column depicts the original transformation in terms of the geometric model, as well as the matrix representing it. The second column shows the same calculations divided in two QPPs, where each QPP executes the operation for 2 amplitudes of the state vector. Granularity is represented by the tuple \((2, 4)\), i.e. 2 rows are being updated and 4 amplitudes from the global state are being read. The last column shows a set of QPPs with granularity \((1, 4)\), which are the most elementary QPPs possible for this example.

QPPs may be associated to nodes in a distributed system but each of them must have an updated copy of the current state vector for the entire system. This implies in the need of an expanded amount of memory in each node as well as heavy communication before each set of operations.
transformations.

4.2. Distribution of Classic Partial Processes
The distribution of Classic Partial Processes (CPPs) [9] allows for the abstraction of information about the columns of the matrix notation in a QT. Partial results are generated from the CPPs for all amplitudes of the resulting state vector.

Figure 3 shows some of the possible distribution of CPPs for the Hadamard operator $H^\otimes2$. The $\perp$ value is generated in the columns that will be omitted according to the proposed abstraction.

In an analogous way to the described in Figure 2, the process depicted in Figure 3 presents its coarser granularity defined by $(4,1)$, and each synchronization will generate a partial amplitude for each state in the computational base. The computation of each CPP is independent from the others and may be executed in parallel if there are enough resources to keep more than one partial amplitude.
The CPPs can also be associated to nodes in a distributed system, where each node receives part of the input state. With this information, it is possible to start calculating CPPs associated to a QT. However, each CPP generates an entire output state vector, with partial values for the results, which increases memory consumption in the nodes.

### 4.3. Mixed Distribution

Both QPP Distribution and CPP Distribution have their advantages and disadvantages. The former requires the storage of only a few amplitudes, at the cost of having the entire current state space for the computation of its subset of amplitudes. The latter does not demand all amplitudes of the current state vector, although an equivalent memory space is necessary to store the partial amplitudes calculated by each CPP.

Our proposed combination of the QPP and CPP distributions in a mixed layout addresses both challenges: managing the communication overhead and reducing the memory usage. This enhanced layout is named **Mixed Partial Process (MPP)**.

The notion of partial process corresponding to a MPP deals with the partition of read/write memory values considering a pair of parameters \((R, W)\). Thus, the representation of a QT demands \(R \times W\) MPPs, with \(R\) and \(W\) indicating the partition number of the read and write memory, respectively.

By the definition of a MPP, it is possible to integrate both interpretations:

1. **(i)** the QPP behaviour, by using a \((1, W)\) MPP configuration where only the write memory partition is considered; and
2. **(ii)** the CPP behaviour, by using a \((R, 1)\) MPP configuration where only the read memory partition is considered.

With the modeling and implementation of MPPs in VirD-GM, programmers receive more control of the memory usage and the granularity of computation. The integration of these features with the VPE-qGM interfaces provides for a more interactive experience with the simulations in the D-GM environment. Although the environment is still much higher-level than the interfaces provided by MPI and CUDA, the programmer retains the capability of fine-tuning the amount of calculations and memory usage according to resources available in the processing nodes.

Figure 4 shows an example of computation layouts in the MPP approach for a Hadamard \(H^{\otimes 2}\) operator. In this context, the application of the mixed reduction in the associated matrix will reduce granularity from \((4, 4)\) to \((2, 2)\) in the distribution of the corresponding computation.

Figure 5 depicts the distribution of the computation among nodes:

1. **(i)** **Node 1** executes a calculation over half the amplitudes from the state vector, thus generating the partial amplitudes for the corresponding positions in the state vector; meanwhile,
2. **(ii)** **Node 2** executes the complementary calculation.

In the end, the partial amplitudes from each state are merged in the server node to obtain the final state for the quantum system using operators such as composition, sum, and others.

### 5. VirD-GM and implementations

The VirD-GM environment is being developed in order to produce the transparent management of distributed execution in the VPE-qGM. This section describes the control-operation of the VirD-GM environment to support the mixed modeling computing and implementations of MPPs.

Distributed simulation of a quantum algorithm from MPPs consists of sending every MPP to a **VirD-Client**, passing the parameters of the MPP and the necessary portion of the current
memory. The VirD-Client performs the computation and generates a list with the calculated partial values, which is then sent back to the VirdServer. Afterwards, the VirdServer updates its memory with the data in this list.

This process is executed for all temporary files generated in the simulation, requiring a file for each step of the simulation. After all steps have been performed, the results can be exported to the VPE-qGM, which can load the simulation results and present them at the graphical interfaces of its editors.

5.1. Implementation

This implementation aims to contribute to reducing spatial and time complexity in multi qubits simulations.
The strategy adopted to reduce the spatial complexity is based on that an MPP may be defined from basic matrices of lowest order, which can be dynamically combined to generate the elements corresponding to the resulting matrix of the tensor product. The rows and columns of a matrix notation in a QT modeled by MPPs define which state vector amplitudes are necessary to perform the computation and also which partial state will be generated from its execution.

Complementary MPPs (interpreting disjoint sets of rows and columns of the same matrix associated with a QT) can be grouped in the descriptor file processes in order to compute an entire QT.

To reduce time complexity, all computation involving input positions with zero or QT elements with zero are not executed because they do not change the final result. In order to know beforehand if a position is associated to a valid iteration (generate a non-zero value), logic operations are carried out using two masks, \texttt{ZERO} and \texttt{ONE}, so there is no need to iterate until the point at which it becomes invalid to discover this.

The main changes that made possible the extension of VirD-GM environment to support these optimizations are briefly described next:

(i) VirD-Loader module: modified to load the simulation memory in a format compatible for communication with the GPU, since Java provides no primitive complex type.
(ii) Process descriptor file: modified for recognition and interpretation of MPPs.
(iii) Data structures: transformed \texttt{Complex} structures into \texttt{Float} structures so that the kernel can interpret two consecutive floating-point numbers as a single complex number.
(iv) VirD-Exec module: modified to send to a VirD-Client only the portion of memory the MPP requires for simulation.
(v) QGM-Analyzer library: responsible for most of the execution, modified to enable calling the CUDA kernel when requested by management to run on GPUs. The framework JCuda [16] is used for kernel invocation and communication with the GPUs.
(vi) CUDA kernel: modeling and implementation of a new CUDA Kernel, providing support to the new optimizations.

6. MPPs Parallel Execution
This section introduces the efforts which enable the distributed/parallel execution of MPPs using the VirD-GM environment.

6.1. Data structure
In order to increase the granularity of distributed computations, the MPPs are defined from basic matrices of smaller order, which are combined in an iterative manner to dynamically generate the elements corresponding to the resulting matrix of the tensor product. When executing an MPP, the runtime uses the following parameters:

(i) Matrices List: list of basic matrices generated by host-code;
(ii) Last Matrix: contains the data of the last basic matrix, and is stored separately from the others because it is constantly accessed.
(iii) Zero: contains the values of each row related to column Zero of each quantum transformation operators used to generate the mask Zero;
(iv) One: contains the values of each line related to column One of each quantum transformation operators used to generate the mask One;
(v) Begin: stores in which position of the matrices list is the first element of each matrix.
(vi) Dimension: stores the dimension of each basic matrix.
Exp: stores the dimension log of each basic matrix.

All values corresponding to a QT matrix are stored in a contiguous data vector and the information about the beginning of the matrices indicate which data belong to which matrix. Moreover, the way these are combined is detailed in the CUDA kernel presented in subsection 6.3.

6.2. Data Allocation on GPUs

The arrangement of data over the different GPU memory spaces is particularly important to ensure good performance. The MPP parameters do not change during the simulation and are common to all threads. Thus, the allocation of that data in the constant memory space becomes an appropriate option. Data copies are made using the JCuda framework.

In addition to the MPP data, three vectors are found in the host memory space: (1) one modeling the current state of the quantum system; (2) one stating for each position which is the next with a non-zero value; and (3) other modeling the next state, which will be calculated. Before the implementation of the CUDA kernel may start execution, both vectors are copied to the GPU global memory.

6.3. CUDA Kernel

The CUDA kernel is implemented in an iterative fashion, since most of the GPUs do not support recursion (only GPUs with Compute Capability 3.5 or higher support recursion). As the kernel described here presents a behavior similar to the tensor product it is able to act on an arbitrary number of basic matrices. Each CUDA thread maintains internal information and execution controls to set access limits in each array.

The CUDA kernel computation can be divided into eight steps, as described below.

Step 1: Constant information initialization, which are common to all the CUDA threads of the application. BITS equivalent to 2 raised to the power of number of MPP qubits, is used in calculation operations of the element position on matrix; AMP informs how many amplitudes each CUDA thread calculates; SHIFT_READ and SHIFT_WRITE define the MPP reading and writing position basis, respectively; TAM_BLOCK sets the block size; TOTAL_ELEMENTS, LAST_MATRIX_ELEMENTS, STACK_SIZE and STACK_LINES are the constant memory vectors parameters.

These constants are defined at run time by changing the kernel base file, where these values are not defined yet because this information varies from execution to execution according to the MPP to be calculated, after that it is possible to compile the resulting kernel.

Step 2: Initialization of each thread’s amplitude vector and local variables, depicted in Figure 6. The only value that differs among them is the rowId, which defines the thread row set.

```c
for (l = 0; l < AMP; l++)
    newAmplitudes[l] = make_cuFloatComplex(0,0);

cuFloatComplex value = make_cuFloatComplex(1,0);
rowId = (TAM_BLOCK * blockIdx.x + threadIdx.x)*dimensionC[STACK_SIZE] + SHIFT_WRITE;
d = dim = prev_pos = pos = zero = one = 0;
```

**Figure 6.** Step 2: Initialization code executed by all threads.

Step 3: Creation of masks used to see if a position has a valid iteration in the thread row set, as shown in Figure 7.
for (ind = 0; ind < (STACK_SIZE); ind++){
    mask = dimensionC[ind] -1;
    d += expC[ind];
    l = (rowId >> (BITS - d)) & mask;
    zero = (zero << expC[ind]) | (rowZeroC[dim + l]);
    one = (one << expC[ind]) | (rowOneC[dim + l]);
    dim += dimensionC[ind];
}
zero = (zero << LAST_EXP) | rowZeroC[dim];
one = (one << LAST_EXP) | rowOneC[dim];

Figure 7. Step 3: Code for masks.

Step 4: Advance in the input vector performing the calculations required for the positions that are different from zero and have a valid iteration (Figure 8).

p = positions[p];
while (p < SIZE){
    temp = readMemory[p];
    pos = p + SHIFT_READ;
    if (((zero & ~pos) | (one & pos))){
        ...
    }
}

Figure 8. Step 4: Calculation of valid iterations.

Step 5: Backtrack the necessary iterations. Based on the previous and current position, the variable used on the iterations has its value regressed to the point in common between the two positions (Figure 9).

back = (pos & (~prev_pos)) >> LAST_EXP;
while (back != 0){
    ind--;
    mask = dimensionC[ind] -1;
    l = (rowId >> (BITS - d)) & mask;
    c = (prev_pos >> (BITS - d)) & mask;
    value = cuCdivf(value, matricesC[initMatrixC[ind] + l*dimensionC[ind] + c]);
    d -= expC[ind];
    back = back >> expC[ind];
}

Figure 9. Step 5: Backtracking a variable to a common point.

Step 6: Advance in matrix iteratively, calculating the multiplication between the elements indexed according to the rowId and current position values (Figure 10).

for ( ; ind < (STACK_SIZE); ind++){
    mask = dimensionC[ind] -1;
    d += expC[ind];
    l = (rowId >> (BITS - d)) & mask;
    c = (pos >> (BITS - d)) & mask;
    value = cuCmulf(value, matricesC[initMatrixC[ind] + l*dimensionC[ind] + c]);
}

Figure 10. Step 6: Calculation of the multiplication of elements indexed by rowId.
Step 7: **Perform the new amplitudes’ partial update** using the value calculated on the previous step and going through the last matrix (Figure 11).

```c
    c = pos & (AMP - 1);
    temp = cuCmulf(value, temp);
    for (l = 0; l < AMP; l++)
        newAmplitudes[l] = cuCaddf(newAmplitudes[l], cuCmulf(temp, lastMatrixC[l*AMP + c]));
    prev_pos = pos;
}
p = positions[p+1];
```

**Figure 11.** Step 7: Partial update of amplitudes.

Step 8: **Copy of vector data** with the new amplitudes calculated to the write state vector in GPU global memory (Figure 12)

```c
    rowId -= SHIFT_WRITE;
    for (l=0; l<AMP; l++)
        writeMemory[rowId+l] = newAmplitudes[l];
```

**Figure 12.** Step 8: Updating the global memory with the new amplitudes.

All these steps define the CUDA kernel and what each CUDA thread will execute recalculating \( n \) amplitudes of the new state vector according to the QTs applied, where \( n \) is equivalent to the size of the last matrix.

7. Evaluation
For validation and performance analysis of the simulation of quantum algorithms from MPPs in the VirD-GM, case studies with QTs Hadamard up to 21 qubits \( (H^{\otimes 18}, H^{\otimes 19}, H^{\otimes 20} \text{ and } H^{\otimes 21}) \) were considered.

The operator Hadamard was chosen because it corresponds to a QT (associated with dense matrix) that has the highest execution cost in the simulation environment.

The methodology adopted for the simulation distributed via GPU considers running 10 simulations of each instance of the operator Hadamard, considering each of the settings within the cluster of GPUs and three types of input state memory, the number of MPPs never exceeds the number of GPUs.

The tests were performed in four desktops with Intel Core i7 processors and 8 GB of RAM memory. Two desktops had NVIDIA GT640 GPUs, while the remaining two had NVIDIA GTX560 GPUs. An additional machine, connected to the cluster through a Fast Ethernet was required to run the VirD-GM server. As software components, we have: JCUDA 0.5.5, NVIDIA CUDA Toolkit 5.5, and Ubuntu 12.04 64 bits. For simulations with 1 and 2 clients, the two desktops used were those with NVIDIA GT640.

The number of MPPs in each simulation is equal to the number of execution clients. Based on the pattern described in Section 4.3, related configurations were used as in the following:

(i) 1 VirD-Client: \( (1 - 1) \);
(ii) 2 VirD-Client: \( (1 - 2), (2 - 1) \);
(iii) 4 VirD-Client: \( (1 - 4), (4 - 1), (2 - 2) \).
Moreover, the memories used are reported in the following:

(i) Memory 1: all positions with non-zero values (superposition state);
(ii) Memory 2: non-zero and zero values alternated (intermediate state);
(iii) Memory 3: first position with a non-zero value and the rest with zeros values (classical state);

7.1. Results

Simulation times obtained can be seen in Tables 1, 2 and 3. In the simulations with memory 1 and 2, the standard deviation maximum of 0.79% was measured in the $H^{18}$ with configuration 4 – 1 using memory 2. In the simulations with memory 3, the standard deviation maximum of 4.9% was measured in the $H^{20}$ with configuration 1 – 4.

| Table 1. Simulation average times obtained for memory 1, measured in seconds. |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| QT | Conf. | VirD-Client | VirD-Clients | VirD-Clients |
|----|-------|-------------|--------------|--------------|
| $H^{18}$ | 1 – 1 | 8.83 | 5.11 | 5.15 | 3.28 | 3.41 | 3.27 |
| $H^{19}$ | 1 – 2 | 31.03 | 16.43 | 16.70 | 9.11 | 9.32 | 9.25 |
| $H^{20}$ | 2 – 1 | 119.74 | 63.89 | 63.34 | 33.30 | 33.70 | 33.34 |
| $H^{21}$ | 4 – 1 | 470.80 | 248.96 | 245.47 | 126.62 | 125.78 | 127.02 |

| Table 2. Simulation average times obtained for memory 2, measured in seconds. |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| QT | Conf. | VirD-Client | VirD-Clients | VirD-Clients |
|----|-------|-------------|--------------|--------------|
| $H^{18}$ | 1 – 1 | 6.19 | 3.77 | 3.79 | 2.56 | 2.64 | 2.56 |
| $H^{19}$ | 1 – 2 | 20.95 | 11.27 | 11.43 | 6.43 | 6.60 | 6.52 |
| $H^{20}$ | 2 – 1 | 78.75 | 41.21 | 41.33 | 21.74 | 22.22 | 21.78 |
| $H^{21}$ | 4 – 1 | 308.19 | 158.79 | 158.24 | 81.16 | 81.70 | 81.52 |

| Table 3. Simulation average times obtained for memory 3, measured in seconds. |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| QT | Conf. | VirD-Client | VirD-Clients | VirD-Clients |
|----|-------|-------------|--------------|--------------|
| $H^{18}$ | 1 – 1 | 1.24 | 1.25 | 1.25 | 1.30 | 1.30 | 1.28 |
| $H^{19}$ | 1 – 2 | 1.48 | 1.50 | 1.49 | 1.68 | 1.56 | 1.53 |
| $H^{20}$ | 2 – 1 | 1.93 | 1.96 | 1.96 | 2.23 | 2.15 | 2.06 |
| $H^{21}$ | 4 – 1 | 2.86 | 2.95 | 2.92 | 3.31 | 3.30 | 3.18 |

Figures 13, 14 and 15 present relative speedups obtained for the different types of configurations with the baseline being the 1 – 1 configuration, which uses 1 VirD-Client. The relative speedup is the ratio between the baseline execution time and the parallel one:

$$Speedup_{parallel} = \frac{t_{baseline}}{t_{parallel}}$$

It can be observed that there is performance gain when increasing the number of clients for memories 1 and 2, regardless of the configuration used. For these cases, speedups tend to the ideal with the increased number of QTs qubits. This is because the time required
for communication between the server and the client becomes less significant compared to the running time on the client.

Tests with the memory 3 (Figure 15) showed no performance gain because the time spent running is small, since there is only one valid position for computation, then the time spent on communication becomes the simulation bottleneck. Notice that this graph presents a different range of values in the vertical axis to better illustrate the differences between bars.

In previous works, the main improvements in the D-GM environment were already introduced, using QPPs in order to control the granularity of computations in both approaches:

(i) distributed computations by using multicore CPU architecture in the VirD-GM [1], obtaining distributed simulation of systems comprised by Hadamard transformations from 14 to 17 qubits); and

![Figure 13. Speedup for Memory 1](image1)

![Figure 14. Speedup for Memory 2](image2)
massive parallel computations, by using multicore parallel/streaming processors provided by GPU architecture, in the extension of the qGM-Analyzer library which support executions in the VPE-qGM [9] (up to 20 qubits for Hadamard transformations performed over single GPU).

Based on these previous results, it is clear that there is a performance improvement obtained by QTs represented by dense matrices (as the Hadamard Transformations) supported by the present GPU approach in the VirD-GM (up to 21 qubits for Hadamard transformations performed over a GPU cluster) in comparison with the distributed simulation supported by multi-core CPU architecture in the VirD-GM.

Other QTs (as controls and measurement operators) are still being incorporated to multi-GPU units approach therefore unavailable in this step for comparative analysis.

8. Conclusion and Final Remarks
The extension of the VirD-GM environment described in this paper provides high performance and scalable simulation of quantum algorithms. We tackled the issues of high spatial complexity and execution time by the use of MPPs in a computing platform comprised of computers with manycore GPUs, integrated through a network.

The concept of a MPP and the read/write memory partition allow the granularity control of an application. Therefore, the GPU memory does not seem an important limitation in the increment of quantum application dimensions, increasing scalability.

As a case study, we simulated the Hadamard Transformation, a challenging quantum algorithm that stresses both spatial and temporal complexities.

The computations optimization for different memory types and the development of the extension to support quantum distributed simulation in GPUs using the VirD-GM environment consist in a solution to reduce simulation time.

Regarding simulations with different types of memories, the results showed that it is possible to reduce the simulation time for memories with zero values, and when the number of non-zero values is small, simulation time can be drastically reduced.

Considering distributed simulations, the results showed that it is possible to obtain performance gain with increasing number of clients, except for extreme cases where the number
of non-zero values in memory is very small. And transformations with different configurations, but using the same number of clients, have a similar simulation time, this means that the programmer is free to define the configuration of the MPPs that looks better according to the resources available, without worrying about having a great loss of performance.

In future works, we intend to develop strategies that take into account both the hardware architectures and the mathematical characteristics of quantum systems in order to further improve performance and the number of qubits that may be simulated in classical computers.

We also intend to test new approaches that may harness the potential of cloud computing for simulation of quantum algorithms, while the extra communication costs are considered.

Acknowledgements
This work is supported by the Brazilian funding agencies CAPES, FAPERGS (PqG 002/2014, process number 309533/2013-9), MCTI/CNPQ (Universal and PQ, process numbers 448766/2014-0 and 309533/2013-9) and by the CNPq/PRONEX/FAPERGS Green Grid project.

References
[1] A. Avila, A. Maron, R. Reiser, and M. Pilla, “Extending the VirD-GM environment for the distributed execution of quantum processes,” in Proc. of the XIII Symposium on Computational Systems – Scientific Initiation Workshop (WSCAD-WIC). Petrópolis: IEEE CPS, 2012, pp. 1–4.
[2] A. d. Avila, A. Maron, R. H. S. Reiser, M. Pilla, and A. Yamin, “GPU-aware distributed quantum simulation,” in Symposium on Applied Computing. Gyeongju: Proc. of the 29th ACM Symposium on Applied Computing (SAC), March 2014, pp. 860–865.
[3] N. Dickson and M. Amin, “Algorithmic approach to adiabatic quantum optimization,” Physical Review, vol. 85, p. 032303, Mar 2012.
[4] N. Dickson, M. Johnson, M. Amin, R. Harris, F. Altmare, A. Berkley, P. Bunyk, J. Cai, E. Chapple, and P. Chavez, “Thermally assisted quantum annealing of a 16-qubit problem,” Nature communications, vol. 4, p. 1903, 2013.
[5] E. Gutierrez, S. Romero, M. Trenas, and E. Zapata, “Quantum computer simulation using the cuda programming model,” Computer Physics Communications, pp. 283–300, 2010.
[6] M. Henkel, “Quantum computer simulation: New world record on JUGENE,” 2010. [Online]. Available: http://www.hpcwire.com/hpcwire/2010-06-28/quantum_computer_simulation_new_world_record_on_jugene.html
[7] M. Hirvensalo, Quantum computing, ser. Natural Computing Series. New York: Springer, 2004. [Online]. Available: http://opac.inria.fr/record=b1117053
[8] S. Imre and F. Balazs, Quantum Computing and Communications - an Engineering Approach. New Jersey: John Wiley and Sons, 2005.
[9] A. K. Maron, R. H. S. Reiser, and M. L. Pilla, “Correlations from conjugate and dual intuitionistic fuzzy triangular norms and conorms,” in CCGRID 2013 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. NY: IEEE, May 2013, pp. 1–8.
[10] A. Maron, R. H. S. Reiser, M. Pilla, and A. Yamin, “Expanding the VPEaqGM environment towards a parallel quantum simulation of quantum processes using GPUs,” CLEI Electronic Journal, vol. 16, no. 3, pp. 1–18, 2013. [Online]. Available: http://www.clei.org/ceiej/papers/v16i3p3.pdf
[11] ——, “Quantum processes: A novel optimization for quantum simulation,” TEMA: Trends in Applied and Computational Mathematics, vol. 14, no. 3, 2013.
[12] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information. Cambridge University Press, 2000.
[13] K. D. Raedt, K. Michielsen, H. D. Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Itō, “Massive parallel quantum computer simulator,” http://arxiv.org/abs/quant-ph/0608239, 2006.
[14] E. Rieffel and W. Polak, “An introduction to quantum computing for non-physicists,” ACM Comput. Surv., vol. 32, no. 3, pp. 300–335, Sep. 2000. [Online]. Available: http://doi.acm.org/10.1145/367701.367709
[15] M. F. Schmalfuss, A. K. Maron, R. H. S. Reiser, and M. L. Pilla, “qGM analyzer: Supporting of quantum computations in the D-GM environment using c++,” in Proc. of the XIII Symposium on Computational Systems – Scientific Initiation Workshop (WSCAD-WIC). Petrópolis: IEEE, 2012, pp. 1–4.
[16] Y. Yonghong, M. Grossman, and V. Sarkar, “Jcuda: A programmer-friendly interface for accelerating java programs with cuda,” in Euro-Par 2009, Delft,Netherlands, 2009, pp. 1–13.