Computer Physics Communications 181 (2010) 283–300

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Quantum computer simulation using the CUDA programming model

Eladio Gutiérrez [*], Sergio Romero, María A. Trenas, Emilio L. Zapata

Department of Computer Architecture, University of Málaga, 29071 Málaga, Spain

ARTICLE INFO

Article history:
Received 25 November 2008
Received in revised form 1 September 2009
Accepted 22 September 2009
Available online 30 September 2009

PACS:
03.67.Ac

Keywords:
Quantum computing
Parallel processing
Graphics Processing Unit (GPU)
Computer Unified Device Architecture (CUDA)
Single Instruction Multiple Threads (SIMT)

ABSTRACT

Quantum computing emerges as a field that captures a great theoretical interest. Its simulation represents a problem with high memory and computational requirements which makes advisable the use of parallel platforms. In this work we deal with the simulation of an ideal quantum computer on the Compute Unified Device Architecture (CUDA), as such a problem can benefit from the high computational capacities of Graphics Processing Units (GPU). After all, modern GPUs are becoming very powerful computational architectures which is causing a growing interest in their application for general purpose. CUDA provides an execution model oriented towards a more general exploitation of the GPU allowing to use it as a massively parallel SIMT (Single-Instruction Multiple-Thread) multiprocessor. A simulator that takes into account memory reference locality issues is proposed, showing that the challenge of achieving a high performance depends strongly on the explicit exploitation of memory hierarchy. Several strategies have been experimentally evaluated obtaining good performance results in comparison with conventional platforms.

1. Introduction

Not even two decades have barely elapsed since the becoming of quantum computation as a promising discipline. When compared with conventional computation (classic), the so denominated quantum computers are devices that process information on the basis of the laws of the quantum physics and that could solve some problems of nonpolynomic complexity in a much smaller time [14]. Although this is a quite hopeful approach, at the present time certain hard limitations must still be faced. On the one hand, existing technology only allows to construct quantum computers of very reduced dimensions [9], and on the other hand only a very small number of algorithms are known to diminish significantly their complexity when executed in a quantum computer, as they are [6,10,21].

The interest of simulating the behavior of quantum computers is motivated by several factors that go from the purely theoretical, academic and educational ones, to the benchmarking of conventional mono or multiprocessor platforms. Besides, its simulation constitutes a very useful tool in the development and test of new quantum algorithms (at least with data sets of small dimension).

Most of the power of quantum computers is due to the quantum parallelism that allows to perform simultaneous operations on an exponential set of superimposed data. This is why the simulation of this kind of systems requires an exponential effort. Parallelism is a suitable tool in order to mitigate such computational requirements and will allow the emulation of quantum computers of a greater dimension in a reasonable time. The model shown in Fig. 1 is the one followed in this work [18]. The quantum computer acts like an accelerating hardware of the classic processor, which sends the orders required to solve a concrete problem. According to the laws that govern it, it is not possible to know the exact state of this quantum computer. Therefore, the output of the quantum algorithm will be obtained by a process of measurement with certain probability. Such an approach is followed in our parallel simulator, which considers an ideal quantum computer. The interface adopted between the classic computer (host) and the simulation platform is borrowed from libquantum [4], one of the more popular simulation software.

In this paper we show that quantum computers can be efficiently simulated on modern architectures based on Graphic Processing Units (GPU). With this purpose, we propose several approaches to the parallel simulation of the basic operators of an ideal quantum computer framed within the Compute Unified Device Architecture (CUDA) after NVIDIA [16].

2. Related work

The interest of scientists, engineers and academics in quantum computing has given rise to the development of several quantum computer simulators, of which a wide list can be found at [19].

[*] Corresponding author.
E-mail addresses: eladio@uma.es (E. Gutiérrez), sromero@uma.es (S. Romero), matrenas@uma.es (M.A. Trenas), ezapata@ac.uma.es (E.L. Zapata).

0010-4655/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2009.09.021

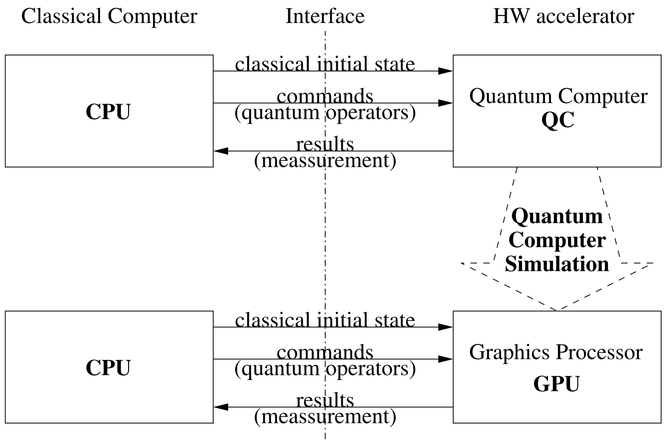


Fig. 1. Quantum computer model as a hardware accelerator and its simulation.

They cover a large number of programming languages and most of them are oriented to instructional or demonstration purposes. Among them we can highlight libquantum [4] a C-based library that allows to write "quantum programs" making use of function calls to elementary quantum gates. The libquantum is a popular library distributed in source format and whose binaries are available in some Linux distributions. In addition it takes part of the benchmark SPEC CPU2006 [22].

Efforts to cope with the high memory and computational requirements can be found in literature. As hardware solutions, several systems based on FPGAs has been proposed as hardware accelerators for mayor quantum computing algorithms, like [1,8,12,13]. On the software side, massive parallel platforms provide high computational resources in order to run faster simulations for a high number of qubits. The main challenge to overcome is how to distribute/communicate efficiently data among processors.

In [17] parallel simulations were carried out on a CrayT3E and an IBM SP2 with MPI message passing library in C, reaching up to 4096 processors with a maximum of 28 qubits. In [15], parallel simulations were executed on a UltraSparcII-based SunEnterprise 4500, using the Solaris thread library, reaching up to 8 threads with a maximum size of 30 qubits. Parallel simulations in [9] were carried out on a Beowulf cluster of 16 nodes based on Pentium 4 processors by using a message passing model allowing to run simulations up to 29 qubits. Simulations proposed in [2] run over an IBM P690 cluster by means of a hybrid model OpenMP (shared memory)/MPI (message passing) with a maximum of 1024 threads achieving executions up to 37 qubits. More recently, in [20], several simulations are tested and compared on massively parallel platforms including IBM Blue Gene/L, IBM Regatta P690, Hitachi SR11000, Cray X1E and others. Programs were coded in Fortran 90 with the MPI library, providing executions with 4096 threads with sizes up to 36 qubits.

3. Quantum computing fundamentals

The ideal quantum computer to be simulated follows the model presented in [5], consisting on the successive application of quantum gates (gate network) to a register with a classical initial state. The quantum bit (qubit) can be imagined as the linear superposition of two homologous classical states denoted as $|0\rangle$, $|1\rangle$, in Dirac notation. Its state can be represented using a complex two-dimensional vector by $\Psi = \alpha_0|0\rangle + \alpha_1|1\rangle$, where the coefficients, or amplitudes, verify $|\alpha_0|^2 + |\alpha_1|^2 = 1$. $|\alpha_0|^2$ and $|\alpha_1|^2$ are interpreted as the probability of measuring $|0\rangle$ or $|1\rangle$. In vector notation, $\Psi = \binom{\alpha_1}{\alpha_0}$, being $|0\rangle = \binom{0}{1}$, $|1\rangle = \binom{1}{0}$.

A quantum register generalizes the qubit definition. The state of an n-qubit quantum register is determined by the linear superposition of the 2^n possible classical states provided by n bits. After this, the state of a quantum register can be written as $\Psi = \sum_{i=0}^{2^n-1} \alpha_i|i\rangle$ with $\alpha_i \in \mathbb{C}$, $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$, where $|\alpha_i|^2$ is interpreted as the probability of obtaining $|i\rangle$ when the register is measured.

Let Ψ be an element of a 2^n-dimensional complex vector space, where $|i\rangle$ constitutes a basis, with $0 \leqslant i \leqslant 2^n - 1$. For example, for $n = 3$, we will write $|6\rangle = |110\rangle = (0\ 0\ 0\ 0\ 0\ 0\ 1\ 0)^T$. By using the Kronecker's tensor product, the elements of the state space basis can be represented by a function of the individual states of the qubits. For example $|6\rangle = |110\rangle = |1\rangle \otimes |1\rangle \otimes |0\rangle$.

The state of a quantum register will evolve according to a transformation, which can be interpreted as an operator U applied to the register state. Quantum physics laws settle that operator U must be a linear and unitary one. It follows that for an n-qubit register, a $(2^n \times 2^n)$-size matrix can be found verifying $UU^* = I$, where U^* is matrix U conjugated and transposed, and I is the unitary matrix. As a consequence, every valid transformation must be reversible. Usually, this kind of transformations are represented as in Fig. 2(a).

Let us consider the application of a transformation over one particular bit, (Fig. 2(b)). In this case, the global transformation will be the tensor product of all the 1-qubit transformations simultaneously applied to each individual qubit. If the 1-qubit operator U is applied to the k-th qubit then:

$$\begin{aligned} U_g &= I \otimes \overset{(n-k-1\ \text{times})}{\cdots} \otimes I \otimes U \otimes I \otimes \overset{(k\ \text{times})}{\cdots} \otimes I \\ &= I^{\otimes n-k-1} \otimes U \otimes I^{\otimes k}. \end{aligned} \tag{1}$$

The transformation applied to one single qubit is given by an unitary quantum gate of order 2×2. Table 1 presents several well-known transformations. The generalization to gates with more than one qubit is straightforward, resulting in an associated matrix of order $2^n \times 2^n$, for n qubits. Notice that the number of qubits at the gate's output must be equal to the one at its input, as it is a reversible transformation. This does not occur with conventional logic. Different minimal universal sets of gates have been proposed, in such a way that any n-qubit transformation can be expressed as a network of these gates. It is proven that no universal minimal set of 1-qubit gates exists. Nevertheless a complete set

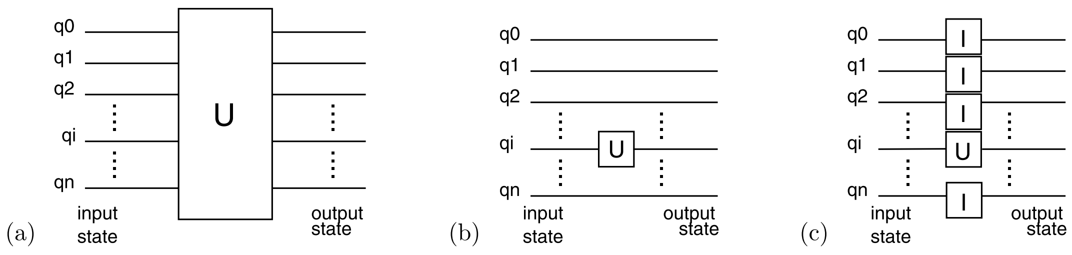


Fig. 2. Representation of quantum transformations as quantum gate networks.

Table 1
Inventory of quantum computer gates.

1-qubit transformations		
Identity	$I = \lvert 0\rangle\langle 0\rvert + \lvert 1\rangle\langle 1\rvert$	I
Pauli X	$X = \lvert 0\rangle\langle 1\rvert + \lvert 1\rangle\langle 0\rvert$	X
Pauli Y	$Y = j\lvert 1\rangle\langle 0\rvert - j\lvert 0\rangle\langle 1\rvert$	Y
Pauli Z	$Z = \lvert 0\rangle\langle 0\rvert - \lvert 1\rangle\langle 1\rvert$	Z
Hadamard	$H = \frac{1}{\sqrt{2}}(X + Z)$	H
y-axis rotation	$R_y(\theta) = \cos(\frac{\theta}{2})I + \sin(\frac{\theta}{2})Y$	$R_y(\theta)$
z-axis rotation	$R_z(\alpha) = e^{j\alpha/2}\lvert 0\rangle\langle 0\rvert + e^{-j\alpha/2}\lvert 1\rangle\langle 1\rvert$	$R_z(\alpha)$
Phase shift	$\Phi(\delta) = e^{j\delta}\lvert 0\rangle\langle 0\rvert + e^{j\delta}\lvert 1\rangle\langle 1\rvert$	$\Phi(\delta)$
2-qubit transformations		
Controlled NOT	$\mathrm{CNOT} = \lvert 0\rangle\langle 0\rvert + \lvert 1\rangle\langle 1\rvert + \lvert 2\rangle\langle 3\rvert + \lvert 3\rangle\langle 2\rvert$	⊕
Controlled Phase Shift	$\mathrm{CPh(K)} = \lvert 0\rangle\langle 0\rvert + \lvert 1\rangle\langle 1\rvert + \lvert 2\rangle\langle 2\rvert + e^{j\frac{2\pi}{K}}\lvert 3\rangle\langle 3\rvert$	K

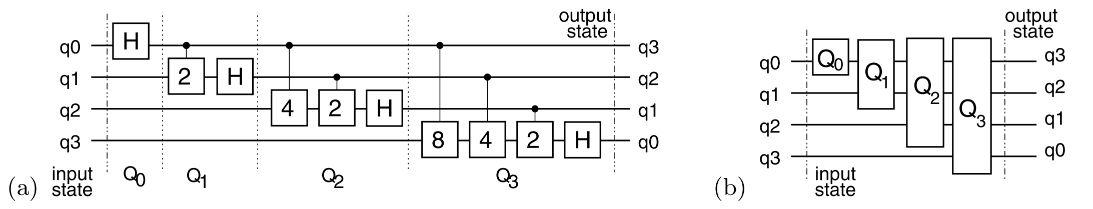


Fig. 3. (a) A QFT implementation using Hadamard (1-qubit) and controlled phase shift (2-qubit) gates. (b) QFT decomposition into steps.

can be built from gates $\Phi(\delta)$, $R_z(\alpha)$, $R_y(\theta)$ and CNOT (2 qubits), as stated in [3].

A remarkable transformation is the Quantum Fourier Transform (QFT) [14] which is a key element of some quantum algorithms like the integer factorization proposed by Shor [21], of great interest in cryptography. An implementation of the QFT is depicted in Fig. 3 in terms of 1-qubit Hadamard gates and 2-qubit controlled-phase gates. As its analogous Fourier Transform, QFT is defined by:

$$\left|\Psi^{\mathrm{out}}\right\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} \alpha_k^{\mathrm{in}} \sum_{c=0}^{2^n-1} e^{\frac{2\pi ck}{2^n} i} |c\rangle. \tag{2}$$

4. The Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA$^{\mathrm{TM}}$) has been introduced by NVIDIA as both a hardware and software architecture for issuing and managing computations on their most recent GPU families, making it to operate as a truly generic data-parallel computing device. An extension to the C programming language is provided in order to develop source codes.

From the hardware point of view, the GPU device consists of an array of SIMT (Single Instruction Multiple Threads) multiprocessors (SM) each one containing several stream processors (SP), as depicted in Fig. 4 [7,16]. Different memory spaces are available. The global device memory is a unique space accessible by all multiprocessors, acting as the main device memory with a large capacity. Besides, each multiprocessor owns a private on-chip memory, called shared memory or parallel data cache, of a smaller size and lower access latency than the global memory. In addition, there are other addressing spaces, omitted in the figure, for specific purposes: texture and constant memories.

CUDA execution model is based on a hierarchy of abstraction layers: grids, blocks, warps and threads as shown in Fig. 5. The thread is the basic execution unit that is actually mapped onto one processor. A block is a batch of threads cooperating together on one multiprocessor and therefore all threads in a block share

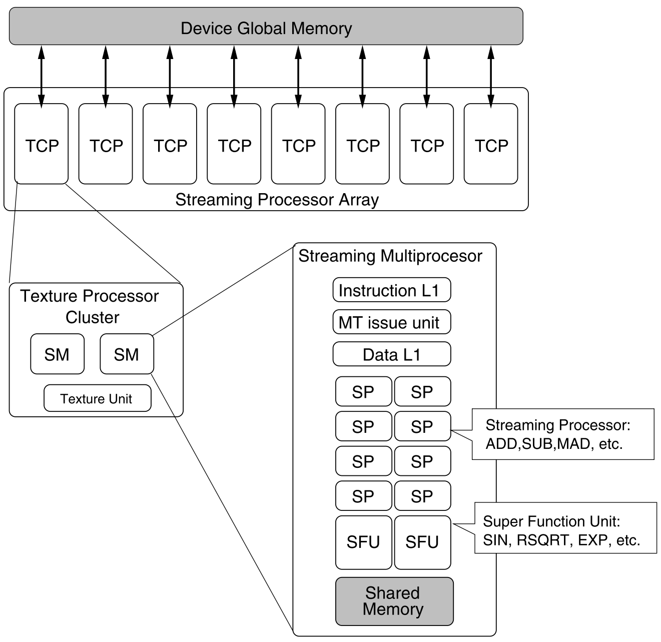


Fig. 4. Organization of processors and memory spaces in CUDA.

the parallel data cache. A grid is composed by several blocks, and because there can be more blocks than multiprocessors, different blocks of a grid are scheduled among the set of multiprocessors. In turn, a warp is a group of threads executing in an SIMT way, so threads of a same block are scheduled in a given multiprocessor warp by warp. In a given moment threads of several blocks may be under scheduling on the same multiprocessor. These are called active blocks (active threads). This feature allows to give a chance for hiding memory latency, as far as warps of active threads can be dispatched while other active ones are waiting for global memory accesses.

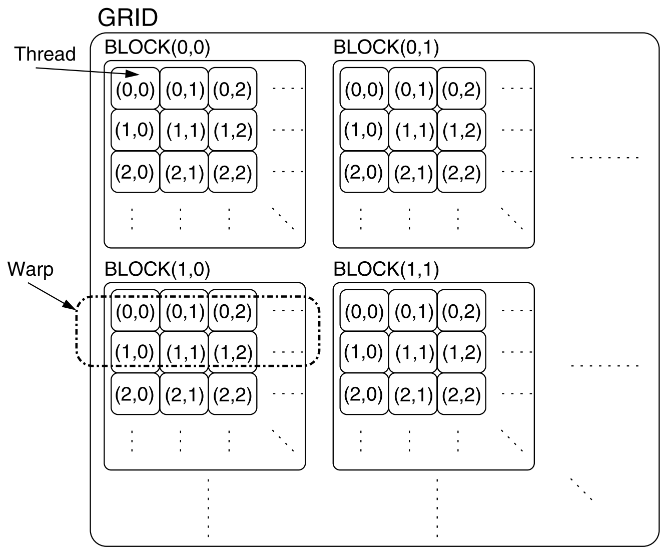


Fig. 5. SIMT execution scheme in CUDA.

Two kinds of codes are considered: those executed by the CPU (host side) and those executed by the GPU, called kernel codes. The CPU is responsible of transferring data between host and device memories as well as invoking the kernel code, setting the grid and block dimensions. Such kernels are intended to be executed in an SIMT fashion over the processors A simple example is shown in Fig. 6 where a floating point array is initialized in the device. After allocating the array space on the global device memory, the kernel code is invoked by the host. The kernel function is executed by the GPU in a SIMT way: each thread will update an element of the global array.

The programmer defines the number of blocks, the number of threads per block and their distribution (1D, 2D, 3D). Nevertheless the dimensionality of warps is a scheduling parameter fixed by the system. The programmer is responsible of applying coding strategies resulting in a good scheduling and workload balancing. In addition, some limitations exist, such as the maximum number of threads per block and the maximum number of blocks in each dimension. Memory accesses and synchronization scheme are other major aspects to take into account. Warp addresses issued by SIMT memory access instructions may be grouped thus obtaining a high memory bandwidth. This is known as coalescing condition that states an optimum transfer speed when consecutive memory locations are accessed by consecutive threads [11,16]. Otherwise, access will be serialized and the resulting latency will be difficult to hide with the execution of other warps of the same block. Global synchronization is not provided at the device side, only threads in a block can be waiting one to each other. Consequently block synchronization mechanism must be explicitly implemented by the host through consecutive kernel invocations.

5. Data parallel simulation

In this section we will describe the computational model of the parallel simulation, based on the CUDA architecture. We will analyze the existent data dependencies following a data-parallel approach, as they will be the main limit to the exploitable parallelism. Hereafter, we will denote the space of coefficients, as $\mathcal{S} = \{\alpha_0, \alpha_1, \ldots, \alpha_{N-1}\}$,[1] being $N = 2^n$. More precisely, $\mathcal{S}^{\text{in}} = \{\alpha_0^{\text{in}}, \alpha_1^{\text{in}}, \ldots, \alpha_{N-1}^{\text{in}}\}$ is the coefficient space in its initial state, before application of the unitary transformation, and $\mathcal{S}^{\text{out}} = \{\alpha_0^{\text{out}}, \alpha_1^{\text{out}}, \ldots, \alpha_{N-1}^{\text{out}}\}$ is the coefficient space after the transformation.

Our target problem is to compute the output state for an n-qubit register, $|\Psi^{\text{out}}\rangle = \sum_{i=0}^{2^n-1} \alpha_i^{\text{out}} |i\rangle$, starting from the initial state $|\Psi^{\text{in}}\rangle = \sum_{i=0}^{2^n-1} \alpha_i^{\text{in}} |i\rangle$, given a transformation U of size $2^n \times 2^n$ ($|\Psi^{\text{out}}\rangle = U|\Psi^{\text{in}}\rangle$). This involves the computation of the values for the set of coefficients $\mathcal{S}^{\text{out}}$, using as input the set of coefficients $\mathcal{S}^{\text{in}}$. Observe that, in general, the application of such a transformation has a computational complexity of order $\mathcal{O}(2^{2n})$. A first step will be to establish the data dependencies between input and output coefficients. For a message passing architecture, they will determine the communications. In the SIMT model of parallelism, this dependencies affect to the memory access conflicts and the synchronization between different threads. Being conservative and as a trivial case, it can be assumed that all coefficients will be transformed and that each output coefficient will depend on all of the input coefficients. However, as aforementioned, by expressing the transformation as a network of quantum gates [3,5], each one affecting a subset of qubits (Fig. 7(a)), dependencies can be managed at gate level.

Without loss of generality, the application of a gate over m consecutive qubits parting from the qubit i will be considered in order to analyze the data dependencies (Fig. 7(b)) for a register of n qubits ($i + m \leqslant n$). In this case, the resulting global transformation will be expressed as:

$$U_g = I^{\otimes n-m-i} \otimes U \otimes I^{\otimes i}, \tag{3}$$

where the matrix associated to the m qubits to be transformed is $U = \sum_{i=0}^{2^m-1} \sum_{j=0}^{2^m-1} u_{i,j} |i\rangle\langle j|$, with size $2^m \times 2^m$.

Let us consider that the transformation will be applied to an initial classical state (an element of the vectorial space basis) whose binary representation is $\Psi^{\text{in}} = |b_{n-1}b_{n-2}\ldots b_1 b_0\rangle_2$:

$$\begin{aligned}
\Psi^{\text{out}} &= U_g \otimes |b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \\
&= \left(I^{\otimes n-m-i} \otimes U \otimes I^{\otimes i}\right) \otimes |b_{n-1}b_{n-2}\ldots b_0\rangle \\
&= |b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \otimes U|b_{i+m-1}\ldots b_i\rangle \\
&\quad \otimes |b_{i-1}b_{n-2}\ldots b_0\rangle \\
&= \begin{cases}
|b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \otimes \left(\displaystyle\sum_{i=0}^{2^m-1} u_{i,0}|i\rangle\right) \\
\quad \otimes |b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \\
\text{if } b_{i+m-1}\ldots b_i = 0 = 0\ldots 00_{(2}, \\
|b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \otimes \left(\displaystyle\sum_{i=0}^{2^m-1} u_{i,1}|i\rangle\right) \\
\quad \otimes |b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \\
\text{if } b_{i+m-1}\ldots b_i = 1 = 0\ldots 01_{(2}, \\
\vdots \\
|b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \otimes \left(\displaystyle\sum_{i=0}^{2^m-1} u_{i,2^m-1}|i\rangle\right) \\
\quad \otimes |b_{n-1}b_{n-2}\ldots b_{i+m}\rangle \\
\text{if } b_{i+m-1}\ldots b_i = 2^m-1 = 11\ldots 1_{(2}.
\end{cases}
\end{aligned} \tag{4}$$

In general, every state will be a linear superposition of these states, so that the coefficient k of the final state vector, with binary expression $k = b_{n-1}b_{n-2}\ldots b_{0\,2}$, can be written as a function of the initial state vector using Eq. (4):

[1] Actually these α_i represent literals, not the coefficients values. So, the set will contain N elements associated to a known state vector $|\Psi\rangle = (\alpha_0, \alpha_1, \ldots, \alpha_{N-1})$.

Host side (CPU)

```
void main(){
 int dimGrid=nBlocks; /* Number of blocks */
 int dimBlock=N/nBlocks; /* Threads per block */
 float *A, *Ad;
 cudaMalloc(Ad, N);

 kernel<<<dimGrid,dimBlock>>>(Ad);

 A=(float *)malloc(N*sizeof( float ));
 cudaMemcpy(A, Ad, N, cudaMemcpyDeviceToHost);
   ...
}
```

Device side (GPU)

```
__global__ void kernel ( float *Ad){
  /*Thread id in the whole grid*/
  int id=blockIdx.x*blockDim.x+
          threadIdx .x;
  /*The following action is
    executed by threads in a
    SIMT way*/

  Ad[id]=id ?0.0:1.0;

  }
```

Fig. 6. Parallel initialization of the state vector to value $|0\rangle$ at the GPU side.

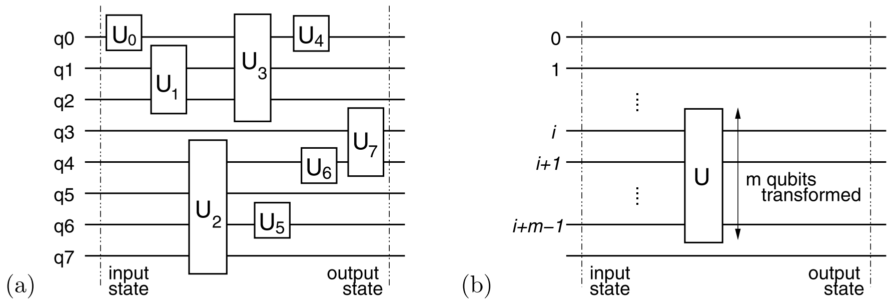


Fig. 7. (a) A quantum computer as a quantum gate network; (b) A gate affecting a subset of qubits.

$$\alpha_k^{\text{out}} = \alpha^{\text{out}}_{b_{n-1}b_{n-2}\ldots b_0} = \begin{cases} \left(\sum_{j=0}^{2^m-1} u_{0,j}\alpha^{\text{in}}_{b_{n-1}b_{n-2}\ldots b_{i+m}|j|b_{i-1}b_{n-2}\ldots b_0}\right) \\ \text{if } b_{i+m-1}\ldots b_i = 0 = 0\ldots 00_{(2}, \\ \left(\sum_{j=0}^{2^m-1} u_{1,j}\alpha^{\text{in}}_{b_{n-1}b_{n-2}\ldots b_{i+m}|j|b_{i-1}b_{n-2}\ldots b_0}\right) \\ \text{if } b_{i+m-1}\ldots b_i = 1 = 0\ldots 01_{(2}, \\ \vdots \\ \left(\sum_{j=0}^{2^m-1} u_{2^m-1,i}\alpha^{\text{in}}_{b_{n-1}b_{n-2}\ldots b_{i+m}|j|b_{i-1}b_{n-2}\ldots b_0}\right) \\ \text{if } b_{i+m-1}\ldots b_i = 2^m-1 = 1\ldots 11_{(2} \end{cases}$$
$$= \sum_{j=0}^{2^m-1} u_{\zeta,j}\alpha^{\text{in}}_{(k\wedge(2^{i+m}-2^i))\oplus 2^j}$$
$$\text{with } \zeta = b_{i+m-1}\ldots b_i = \left\lfloor \frac{k}{2^i}\right\rfloor \bmod 2^m. \tag{5}$$

The binary concatenation of natural numbers is represented as $|$, and it applies to both binary digits and natural numbers, keeping a result of n bits. The bit-wise logical AND is $\wedge$, and $\oplus$ is the also bit-wise exclusive or.

Expression (5) allows to compute the coefficients of the final quantum state. Note that just 2^m input coefficients take part in the computation of each output coefficient, and also, that these 2^m coefficients act as a closed group. This means that each of them only takes part in the computation of the coefficients inside that group, being independent from the rest of groups.

For subsequent use, several definitions are formally introduced. Let us consider a subset of the space of coefficients $\mathcal{T}\subset\mathcal{S}$. We denote the values of the coefficients in $\mathcal{T}$ before and after the application of a certain transformation as $\mathcal{T}^{\text{in}}$, $\mathcal{T}^{\text{out}}$, respectively. For a transformation U_g, we denote the set of output coefficients that are computable using only the coefficients in $\mathcal{T}$ as $U_g(\mathcal{T})$

For example, consider $U_g = I^{n-2}\otimes U\otimes I$, a 1-qubit gate U applied to the qubit 1 of the register. If $\mathcal{T}=\{\alpha_1,\alpha_3\}$, then we will denote $\mathcal{T}^{\text{in}}=\{\alpha_1^{\text{in}},\alpha_3^{\text{in}}\}$ and $\mathcal{T}^{\text{out}}=\{\alpha_1^{\text{out}},\alpha_3^{\text{out}}\}$. From Eq. (5), the computation of coefficients α_1^{out} and α_3^{out} depends on α_1^{in} and α_3^{in}, then, $U_g(\mathcal{T}^{\text{in}})=\{\alpha_1^{\text{out}},\alpha_3^{\text{out}}\}$. It acts as a closed computational set because $\mathcal{T}^{\text{out}}=U_g(\mathcal{T}^{\text{in}})$. However, if $\mathcal{T}=\{\alpha_0,\alpha_4\}$ it will follow that $U_g(\mathcal{T}^{\text{in}})=\emptyset$, as it is not possible to compute any output coefficient with only these two coefficients. Then $\mathcal{T}^{\text{out}}=\{\alpha_0^{\text{out}},\alpha_4^{\text{out}}\}\neq U_g(\mathcal{T}^{\text{in}})$. The following definitions enclose these observations in a more formal style.

Definition 1. Let U_g be a transformation applied to a quantum register. We state that a subset of coefficients $\mathcal{T}\subset\mathcal{S}$ is a closed one when $\mathcal{T}^{\text{out}}=U_g(\mathcal{T}^{\text{in}})$.

Claim 1. *The union of two closed subsets for a certain transformation results in a closed subset.*

Definition 2. A closed subset of coefficients is minimum when, after removal of one of its coefficients, it is not possible to compute any output coefficient for the transformation U_g using the coefficients remaining inside that subset.

Claim 2. *For a certain transformation U_g it is always possible to find a partition of the space of coefficients $\mathcal{S}$ into closed subsets of coefficients.*

Claim 3. *If the transformation only affects m qubits from the total n integrating the register, as the case of Eq. (3), there exist $\frac{2^n}{2^m}$ different minimum closed subsets with cardinality 2^m.*

Observe that there are partitions of $\mathcal{S}$ into closed sets, not necessarily minimum, that can be closed for more that one transformation affecting different qubits. For example, for a 1-qubit gate applied over qubit 0, a partition on minimum closed subsets is: $\{\{\alpha_0,\alpha_1\}\{\alpha_2,\alpha_3\}\{\alpha_4,\alpha_5\}\{\alpha_6,\alpha_7\}\}$, but if the gate is applied to qubit 1, the partition comes from: $\{\{\alpha_0,\alpha_2\}\{\alpha_1,\alpha_3\}\{\alpha_4,\alpha_6\}\{\alpha_5,\alpha_7\}\}$. However, we can find a partition into closed sets, though non-minimum, that is closed both for the 1-qubit transformation applied to qubit 0, and the same transformation applied to qubit 1: $\{\{\alpha_0,\alpha_1,\alpha_2,\alpha_3\},\{\alpha_4,\alpha_5,\alpha_6,\alpha_7\}\}$.

The following two characteristics are of great importance as refers to the locality exploitation on the target architecture.

Definition 3. A closed set $\mathcal{T}\in\mathcal{S}$ is c-coalesced if it can be partitioned into disjoint batches with at least 2^c coefficients in the form: $\alpha_k,\alpha_{k+1},\ldots,\alpha_{k+2^c-1}$, being k a power of 2.

Definition 4. We say that a partition of $\mathcal{S}$ is c-coalesced if all of the subsets taking part into the partition are c-coalesced.

Claim 4. *The union of c-order coalesced closed sets, continues being c-order coalesced.*

A remarkable feature of a partition into closed sets, is that the output coefficients belonging to different closed subsets can be computed in parallel. Next, we study the shape of the minimum closed sets that a certain α_k coefficient belongs to, for different configurations of interest.

5.1. Unitary diagonal transformation

The matrix shape for such a transformation is a diagonal with values $e^{j\phi_i}$. Observe that each coefficient from the state vector is self-transformed independently from the rest. As the minimum closed sets contain just one element, it is possible to find a minimum partition of $\mathcal{S}$ into only one element closed sets. In this case, the coefficient α_k will belong to a minimum set that will be denoted as D_k, given by $D_k=\{\alpha_k\}$. Observe that this partition is valid for any diagonal transformation, or its Kronecker product, as its result will also be diagonal.

5.2. 1-qubit gate applied to only one qubit

Let us consider the application of the 1-qubit U gate, over the qubit i of an n-qubit register, resulting on a global transformation of the form: $U_g=I^{\otimes(n-2-i)}\otimes U\otimes I^{\otimes i}$. From Eq. (5) it can be inferred that the minimum closed set which the coefficient α_k belongs to, is made of two coefficients whose subindexes differ on the bit i. We will denote this set as G_k^i, being:

$$G_k^i=\{\alpha_k,\alpha_{k\oplus 2^i}\}. \tag{6}$$

The partition into closed sets defines an equivalence relation, in which coefficients belonging to the same closed set form an equivalence class, so $G_k^i=G_{k\oplus 2^i}^i$. This way, it is necessary to define its canonical representative, that will be the one with the lowest subscript k of all of the coefficients in G_k^i.

Notice that these minimum closed sets only will be coalesced (order 1) if $i=0$. But it is possible to get partitions of $\mathcal{S}$ into non-minimum closed sets with greater cardinality and a greater degree of coalescing.

Claim 5. *In order to obtain a c-coalesced closed set, for a 1-qubit gate applied to the qubit i, it is necessary to unite at least a number of G_k^i groups given by*:

$$\begin{cases}2^{c-1} & \textit{if } i\leqslant c,\\ 2^c & \textit{if } i>c.\end{cases}$$

Claim 6. *A c-coalesced closed set for a 1-qubit gate applied to the qubit i has a cardinality given by*:

$$\begin{cases}2^{c} & \textit{if } 0\leqslant i\leqslant c-1,\\ 2^{c+1} & \textit{if } i>c.\end{cases}$$

It is possible to generalize expression (6) to a gate applied to more than 1 qubit (qubits $a,b,c,\ldots$). Then, from Eq. (5), it results that the minimum closed set for a coefficient α_k is:

$$G_k^{a,b,c,\ldots}=\bigcup_{u_a=0}^{1}\bigcup_{u_b=0}^{1}\bigcup_{u_c=0}^{1}\cdots\{\alpha_{k\oplus(u_a2^a+u_b2^b+u_c2^c+\cdots)}\}. \tag{7}$$

Corollary 1. *Applying c 1-qubit gates to the c less significant qubits results on a c-coalesced partition of the space of coefficients $\mathcal{S}$ such as*:

$$\{G_0^{0,1,\ldots,c-1},G_{2^c}^{0,1,\ldots,c-1},G_{2\cdot 2^c}^{0,1,\ldots,c-1},G_{3\cdot 2^c}^{0,1,\ldots,c-1},\ldots\}.$$

5.3. Controlled gates

Here, we analyze the transformation $U_g=\left(\begin{array}{cc|c}1&0&\\0&1&\\\hline &&U\end{array}\right)$ corresponding to a controlled gate, being U the matrix associated to a 1-qubit gate. In general, it is deduced from Eq. (5) that the closed set for the coefficient α_k of a generic 2-qubits gate can be used for a controlled gate on qubit j controlled by qubit i:

$$G_k^{i,j}=\{\alpha_k,\alpha_{k\oplus 2^i},\alpha_{k\oplus 2^i},\alpha_{k\oplus 2^{i+j}}\}. \tag{8}$$

Being more precise, the minimum closed set associated with α_k is denoted by:

$$C_k^{i,j}=\begin{cases}D_k^j=\{\alpha_k\} & \text{if } k\vee 2^i\neq k,\\ G_k^j=\{\alpha_k,\alpha_{k\oplus 2^j}\} & \text{if } k\vee 2^i=k.\end{cases} \tag{9}$$

Coefficients α_k with $k\vee 2^i\neq k$ are not transformed, as $\alpha_k^{\text{out}}=\alpha_k^{\text{in}}$. Thus, only half of the coefficients are transformed, with the same pattern as the non-controlled 1-qubit gate U applied to the target qubit. From the viewpoint of coalescing, in order to work over c-coalesced closed groups, it is the target qubit, j, the one to be considered. If $j<c$ the same closed groups can be taken, already coalesced. No operation is necessary over the coefficients that do not transform in these groups. Otherwise, if $j\geqslant c$, it generates the minimum closed sets corresponding to the half of the space that does transform (Eq. (9)), and then, it constructs c-coalesced closed sets by grouping them.

5.4. Gate networks

The simulation of a gate network can be decomposed into a sequence of stages where each stage, U_i, has a lower complexity than the global transformation $U_g=U_1\cdot U_2\cdot U_3\cdots$. Note that because the product of matrices is not a commutative one, a first constraint to the decomposition of a gate network into stages is the order of application. At this point, next definition introduces what will be designated as *simulation stage*. It will be constituted by a sequence of gates from the network, in the correct order, satisfying two features related to the size of the closed sets (cardinality) and the degree of locality (coalescing).

Definition 5. A *simulation stage* for a gate network is defined to be the sequence with more consecutive gates, starting from a given one, in the original order, that allows a partition of the coefficient space $\mathcal{S}$ into groups, that satisfies these three features:

```
BuildSimulationStage(r,c){
    $G_k = \{\}, \ \forall\, 0 \le k < N$;
    $Gates = \{\}$;
    while ( exist_more_gates ()){
        $(i,g)$=get_next_gate (); /* Gate $\#g$ applied over qubit $\#i$ */
        for (k=0; k<$2^n$, k++){
            /* If $0 \le i \le c-1$, a partition of $\mathcal{S}$ can be found into
             *    $c-$coalesced sets $\bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u}$ of cardinality $2^c$
             * If $i \ge c$, a partition of $\mathcal{S}$ can be found into
             *    $c-$coalesced sets $\bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u}$ of cardinality $2^{c+1}$
             */
            $G^i_k = \{\alpha_k, \alpha_{k\oplus 2^i}\}$; /* Minimum closed set for gate $g$ */
            $G^{\text{temp}}_k = G_k$;
            foreach $\alpha_u \in G^{\text{temp}}_k${
                /* $G_k$ is a closed set for this stage including the candidate $g$ */
                $G_k = G_k \bigcup G^i_u$;
            }
            if ( $Card(G_k) > 2^r$ ) return $Gates$;
        }
        $Gates = Gates \bigcup \{g\}$;
        foreach $\alpha_k \in \mathcal{S}$ {
            $P_k = \bigcup_{u=0}^{2^c-1} G_{k\oplus 2^u}$ /* $P_k$ is a $c-$coalesced partition of $\mathcal{S}$*/
        }
    }
    return $Gates$;
}
```

Fig. 8. An algorithm for defining our coalescing-oriented simulation stage.

(i) groups are closed for every gate in the stage;
(ii) the cardinality of all the groups is at most 2^r, for a certain value r;
(iii) all the groups are at least c-coalesced for a certain c.

Algorithm in Fig. 8 provides the set of gates for a simulation stage fulfilling the cardinality and coalescing parameters, r and c. The algorithm begins taking the first network gate, applied to qubit i, whose minimum closed set is expressed in Eq. (6). In each iteration, G_k represents the minimum closed set for the gates being aggregated to such a stage. This set may double its cardinality with each new gate. The algorithm stops with the last gate or when the cardinality of G_k surpass 2^r. As a result, P_k represents a c-coalesced set built by the aggregation of minimum closed sets for all gates in the stage. We will denote each closed set P_k associated to one stage as *coalesced coefficients plane*. In order to illustrate the algorithm several examples follow.

Example 1 *(Applying the algorithm starting with a given gate).* Let's assume stages with parameters c and r ($c < r \leqslant n$). Starting with a 1-qubit gate or a controlled one whose target is qubit i, we find two possibilities:

- If it is $i \geqslant c$, the grouping $P_k = \bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u}$, defined in Fig. 8, is a closed set for the transformation applied to qubit i, which is c-coalesced and with cardinality $2^c + 1$. These groupings make a partition of $\mathcal{S}$ and verify: $P_k = \bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u} = G_k^{0,1,2,\ldots,c-1,i}$.
- On the contrary, if $0 \leqslant i \leqslant c-1$, the grouping $P_k = \bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u}$ also defines a c-coalesced partition, but it has a cardinality 2^c. Then, these groupings make a partition of $\mathcal{S}$ verifying: $P_k = \bigcup_{u=0}^{2^c-1} G^i_{k\oplus 2^u} = G_k^{0,1,2,\ldots,c-1}$.

In the next iteration, when incorporating a new gate applied to qubit j, three cases may occur:

- If $0 \leqslant j \leqslant c-1$, sets P_k are already closed both for i and for j and they will not change. Then, the qubit j can be incorporated into the stage.
- If $j = i$, sets P_k will not change. The gate can be incorporated directly.
- If $j \geqslant c$, $j \neq i$, a set P_k will become either $P_k = G_k^{0,1,2,\ldots,c-1,i,j}$, if $i \geqslant c$, or $P_k = G_k^{0,1,2,\ldots,c-1,j}$, if $i \leqslant c-1$. In any case, this means that the cardinality of these c-coalesced sets is doubled.

In general, insertion of a new gate in the stage affecting qubit i means that the c-coalesced groups will remain the same when one of the following conditions is satisfied: there was already a gate affecting qubit i in the stage, or $0 \leqslant i \leqslant c-1$. If not so, insertion of such a gate involves that the cardinality of the c-coalesced closed groups for the stage is doubled. The next results illustrate this fact.

Claim 7. *The cardinality of the c-coalesced closed groups generated from the algorithm in Fig. 8, for a simulation stage, will be $2^{c+d} \leqslant 2^r$ if such a stage contains:*

(i) *an indeterminate number of gates affecting qubits between 0 and $c-1$, and*
(ii) *gates transforming at most d qubits more significant (strictly) than qubit $(c-1)$.*

Corollary 2. *As the maximum cardinality of the partition of $\mathcal{S}$ into c-coalesced closed groups is required to be at most 2^r, the number d of qubits, more significant (strictly) than qubit $(c-1)$, that can be transformed by the simulation stage, is at most $r-c$, that is, $d \leqslant r-c$.*

Example 2 *(Kronecker-factorizable gate $U^{\otimes n}$).* Let's consider the transformation $U^{\otimes n}$ applied to an n-qubit register, where U is a generic 1-qubit gate. Beginning by the less significant qubit, gates will be aggregated in increasing order. After the first c gates, it will result on a partition of the space of coefficients $\mathcal{S}$ into c-coalesced closed groups with cardinality 2^c: $G_k^{0,1,\ldots,c-1}$. After this point, it is possible to add $r-c$ more gates until the

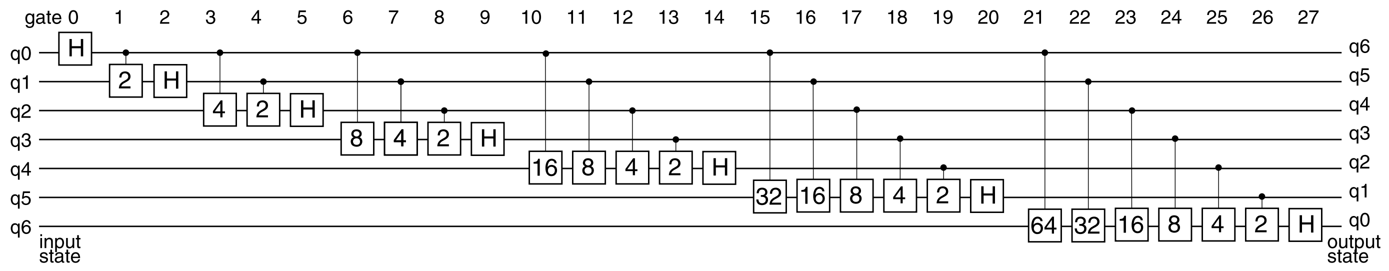


Fig. 9. A QFT implementation for a 7-qubit register.

Table 2
Decomposing of QFT into simulation stages observing coalescing and cardinality parameters $c = 3$, $r = 5$.

Gate	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Control qubit	–	0	–	0	1	–	0	1	2	–	0	1	2	3	–	0	1	2	3	4	–	0	1	2	3	4	5	–
Target qubit	0	1	1	2	2	2	3	3	3	3	4	4	4	4	4	5	5	5	5	5	5	6	6	6	6	6	6	6
$Card(P_k)$	2^3	2^3	2^3	2^3	2^3	2^3	2^4	2^4	2^4	2^4	2^5	2^5	2^5	2^5	2^5	2^4	2^4	2^4	2^4	2^4	2^4	2^5	2^5	2^5	2^5	2^5	2^5	2^5
Stage	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

```
Host                                                  Device ( kernel )

$|\Psi_0^{in}\rangle =$ initial state
for (s=0; s<nStages; s++){                            foreach $j \in \{0, 2^n - 1\}$ {
    $|\Psi^{out}\rangle_i = U_s|\Psi^{in}\rangle_i$ ——————————>    Compute $\alpha_j^{out}$ for transformation $U_s$
    $|\Psi^{in}\rangle_{i+1} = |\Psi^{out}\rangle_i$                }
}
```

Fig. 10. Simulation scheme.

closed groups will reach the highest cardinality (2^r), giving rise to a partition into $G_k^{0,1,\ldots,r-1}$. So, the first stage (stage 0) comprises gates from qubit 0 through qubit $r-1$. The stage p will include those gates applied to qubits from $r+(r-c)(p-1)$ to $r+(r-c)p-1$, corresponding to a partition into closed groups of the form $G_k^{0,1,\ldots,c-1,r+(r-c)(p-1),\ldots,r+(r-c)p-1}$. For $n > r$, the number of stages will follow from the inequality $r+(r-c)(p-1) \leqslant n-1 \leqslant r+p(r-c)-1$. This fact translates into the following result.

Claim 8. *The number of stages produced by algorithm in Fig. 8 for the transformation $U^{\otimes n}$ is:*

$$nStages = \begin{cases} 1, & if\, n \leqslant r, \\ 1 + \lceil \frac{n-r}{r-c} \rceil, & if\, n > r. \end{cases} \tag{10}$$

Note that according to Claim 7 and Corollary 2, for $U^{\otimes n}$, the number of qubits involved in the first stage is at most r, subsequent intermediate stages involve $r-c$, but the last stage may have less than $r-c$ gates.

Example 3 *(Quantum Fourier Transform (QFT)).* This example analyzes the decomposition into stages of the QFT, as implemented in Fig. 9. Unlike previous examples, several controlled gates appear. In general, its minimum closed set will be given by Eq. (8). However, we can consider that the closed groups for controlled gates are equivalent to those of the 1-qubit gate (Eq. (9) applied to the target bit), as refers to its dependency analysis. The control qubit only indicates if the associated computation is performed or not during simulation. Thus, controlled gates require half the computation than the 1-qubit gates with the same pattern of dependencies. Table 2 illustrates the application of the algorithm for particular values $c = 3$, $r = 5$. The resulting number of stages is $\lceil \frac{n-r}{r-c} \rceil + 1 = 2$.

6. Implementation

This section introduces the implementation of the simulator developed using the CUDA programming model. The simulator carries out the computation of the output state of a quantum computer considering a global transformation U_g, as a sequence of stages. The general outline of the simulation is shown in Fig. 10. The outer loop at the host side traverses every stage invoking kernel codes, so that the computation of the output coefficients is done in parallel on the GPU device.

The approach to the kernel is based on considering a partition to the coefficient space into closed sets. Each closed set can be processed independently as it does not depend on any other. The closed sets act as computation units at block level, and they will be assigned to CUDA blocks, so that each block will be in charge of processing one (or several) of such sets. Threads in a block will be responsible for transforming the coefficients from the closed sets in parallel following an SIMT execution. It is necessary to define the granularity of the assignment at two levels. On the one hand it should be defined which are the closed groups assigned to a block. And in turn, within the block, it should be defined how many and which coefficients does each thread process. A thread could go from computing just a couple of coefficients, until the whole closed group. The dimensionality of the grid, that is, the number of blocks and the number of threads per block, will determine this granularity, conditioning aspects like the necessary synchronization.

In our implementation, the whole vector of coefficients resides in device global memory. Nevertheless, as refers to the improvement of the execution time, threads work on local copies of the closed sets placed on the shared memory. The challenge is to find out an assignment function for the threads to the computation space (closed sets) and to the memory spaces (global and shared) where the coefficients are placed. This fact implies the definition of efficient mapping functions between the identifiers of these spaces, as it is shown in Fig. 11. From the performance viewpoint, it is im-

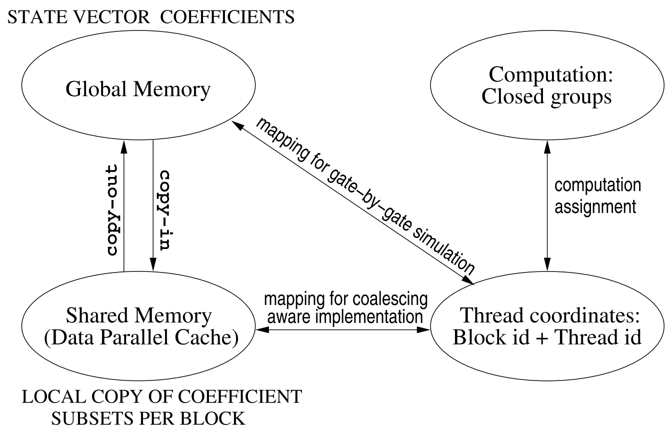

STATE VECTOR COEFFICIENTS
Global Memory
Computation: Closed groups
mapping for gate–by–gate simulation
copy-out
copy-in
computation assignment
Shared Memory (Data Parallel Cache)
mapping for coalescing aware implementation
Thread coordinates: Block id + Thread id
LOCAL COPY OF COEFFICIENT SUBSETS PER BLOCK

Fig. 11. Thread, computation and coefficient spaces.

portant to take into account the peculiar memory features of the CUDA memory model. In order to maintain a good transfer bandwidth and to avoid the serialization of the threads, it is crucial to perform coalesced accesses.

The parallelization strategy proposed in this work will consider stages as defined in Definition 5. This approach exploits the locality and memory hierarchy features, resulting in a good performance. We will designate this implementation as *coalescing-aware*. A particular case is one in which the stage consists of a single gate. This implementation simplifies the program code, at the expense of performance, and will be used just for comparison purposes. This last one will be referred as *gate-by-gate* implementation.

6.1. Coalescing-aware implementation

With the purpose of obtaining a high performance from the memory system, it is necessary to use shared memory as way of a fast memory cache. There is an important difference: data loading is to be made explicitly. From the viewpoint of the kernel code to be executed in the GPU, this approach results in three steps. First, the copying-in of the subset of coefficients to be transferred to the shared memory. This subset has to be a closed one for all the gates in the stage. Second, the gates belonging to the stage will be applied over this subset. This processing will be in-place in the shared memory and a thread-level synchronization between gate transformations is necessary. Finally, the stage ends with the copy-out of the transformed coefficients, that will be returned to global memory. This transformed coefficients go back to their original locations. It will be essential to undertake the copy-in and copy-out operations in a coalesced way.

The way the coefficients are assigned to the CUDA blocks, and how they are copied to the shared memory, are derived from Definition 5. Each one of the resulting sets P_k (Fig. 8) associated to a stage can be transferred in a coalesced way and there are no dependencies between different P_k. This way, each set P_k turns out to be the workload that will be assigned to each CUDA block. Pseudocode of Fig. 12 describes the computations that each CUDA block must carry out. All the computations, including copy-in/out, will be performed in parallel, following the SIMT model.

As an example, let's consider $c = 2$, $r = 4$. A simulation stage adjusting to these characteristics is shown in Fig. 13. The stage affects the qubits from 0 through $c - 1$, and the two more significant qubits, i, j ($c \leqslant i < j$). The algorithm of Fig. 8 allows a partition of the coefficient space into planes P_k, which a given coefficient α_k belongs to. Fig. 13 shows this situation for a given k such that its bits from 0 to $c - 1$, i and j, are zero, that is $k \wedge (2^n - 1 - 2^i - 2^j - \sum_{u=0}^{c-1} 2^u) = k$.

```
Copy_in($P_k$)
synchthreads()
foreach $U \in stage$ {
    $P_k = U(P_k)$
    synchthreads()
}
Copy_out($P_k$)
```

Fig. 12. Kernel pseudocode for the coalescing-aware simulation, at block level.

Sets P_k can be seen as a plane made of $2^c \times 2^{r-c}$ coefficients, inside the volume of all the coefficients (Fig. 14). Each P_k establishes an equivalence class, such that any coefficient is a representative of that class. So, in this example, after $k \wedge (2^n - 1 - \sum_{u=0}^{c-1} 2^u - 2^i - 2^j) = k$ it results that $P_k = P_{k+1} = \cdots = P_{k \oplus 2^i} = \cdots = P_{k \oplus (2^j + 2^i) + 3}$. We can chose the α_k coefficient with lowest value of k as the canonical representative of such a class. This way, the plane P_k that contains the coefficient α_k, will have a canonicalizing function $P_{k_{\text{canonical}}} = P_k$ where $k_{\text{canonical}} = k \wedge (2^n - 1 - \sum_{u=0}^{c-1} 2^u - 2^i - 2^j)$. Generally, the subindex for the canonicalizing function is obtained setting a zero value on bits 0 to $c - 1$, and those ones that being above $c - 1$ have qubits to be transformed by the stage. As the assignment to CUDA blocks is performed at the level of the closed group P_k, the total number of blocks will be the number of planes, 2^{n-r}.

Continuing the example, Fig. 15 illustrates the processing of the plane P_0 by the CUDA block it is assigned to. In the first stage, the coefficients belonging to P_0 are copied-in from global memory to shared memory keeping a coalescing degree 2^c. After this, all the gates of the stage are computed one after the other, working over the shared memory. Each thread will be on charge of computing the transformation for a minimum closed set for that gate, made of just two coefficients, as can be observed in Fig. 15. As a consequence, the number of threads per CUDA block will be half the number of coefficients inside plane P_k, that is, $\frac{2^r}{2}$. Observe that computations are carried out in-place, and a thread level synchronization is required each time a gate transformation is calculated. Finally, in the copy-out, the coefficients already processed will go back to global memory. This transference also verifies the coalescing condition.

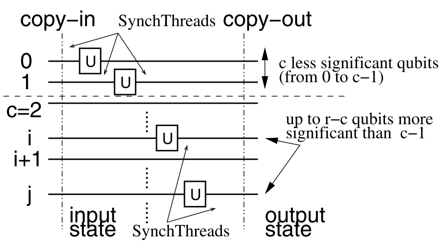
It remains unsolved how to allocate each of the 2^{n-r} planes P_k to the blocks. The canonical representatives, previously defined, have a subscript with zero value on bits 0 to $c - 1$, and also those ones that being above $c - 1$ are transformed on the stage. Thus, the block identifier *blockId* can be mapped to a P_k plane via insertion of zeros in such positions of the binary expression of *blockId*. In a more formal way, the plane corresponding to block *blockId* will be in this example, $P_{\mathit{zero\_xmlitbit\_insert}(\mathit{blockId},\{0,1,\ldots,c,i,j\})}$, where function *zero_bit_insert*[2] inserts zeros in binary positions $\{p_0, p_1, \ldots, p_{M-1}\}$. Thus, if we consider $i = 5$, $j = 6$, the block with identifier $\mathit{blockId} = 11111_2$ will be on charge of computing plane $P_{\mathit{zero\_bit\_insert}(11111_2,\{0,1,5,6\})} = P_{110011100}$.

When there are not enough gates to constitute a plane of 2^r coefficients each block will be on charge of processing several consecutive planes in such a way that the effective coalescing is increased, until the cardinality 2^r is fulfilled. After this, the number of planes to be processed by a block will be $\frac{2^r}{\mathit{Card}(P_k)}$.

6.2. Gate-by-gate implementation

For comparative purposes it is interesting to analyze the case in which each simulation stage consists of a single gate. In this

[2] $\mathit{zero\_bit\_insert}(B, \{p_0, p_1, \ldots, p_{M-1}\}) = \sum_{u=-1}^{M-1} \sum_{v=p_u+1}^{p_{u+1}-1} b_{v-(u+1)} 2^v$ with $p_{-1} = -1$, $p_M = m + M$, $p_0 < p_1 < p_2 < \cdots < p_{M-1}$, for a B with binary expression $B = \sum_{u=0}^{m-1} b_u 2^u$.


copy–in
SynchThreads
copy–out
0
1
c=2
i
i+1
j
U
c less significant qubits (from 0 to c−1)
up to r–c qubits more significant than c−1
input state
SynchThreads
output state

$$G_k^{0,1\ldots c-1}$$

$$P_k = \left\{ \begin{array}{llll} \alpha_k & \alpha_{k+1} & \alpha_{k+2} & \alpha_{k+3} \\ \alpha_{k\oplus 2^i} & \alpha_{k\oplus 2^i+1} & \alpha_{k\oplus 2^i+2} & \alpha_{k\oplus 2^i+3} \\ \alpha_{k\oplus 2^j} & \alpha_{k\oplus 2^j+1} & \alpha_{k\oplus 2^j+2} & \alpha_{k\oplus 2^j+3} \\ \alpha_{k\oplus(2^j+2^i)} & \alpha_{k\oplus(2^j+2^i)+1} & \alpha_{k\oplus(2^j+2^i)+2} & \alpha_{k\oplus(2^j+2^i)+3} \end{array} \right\}$$

$$G_k^{i,j}$$

$$\text{with } k = k \wedge \left(2^n - 1 - \sum_{u=0}^{c-1} 2^u - 2^i - 2^j\right)$$

Fig. 13. Example of coalesced plane associated to a simulation stage with coalescing $c = 2$ cardinality $r = 4$.

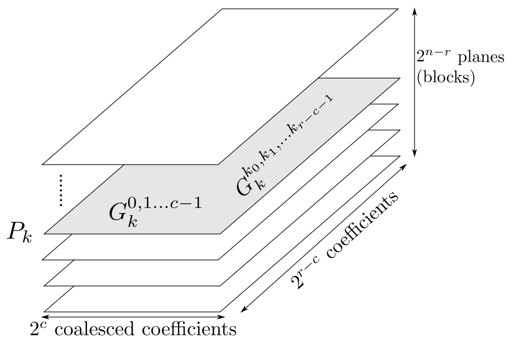

2^{n-r} planes (blocks)
$G_k^{k_0,k_1,\ldots k_{r-c-1}}$
$G_k^{0,1\ldots c-1}$
P_k
2^{r-c} coefficients
2^c coalesced coefficients

Fig. 14. State vector coefficient space arranged in P_k planes.

situation, there won't be any reuse of the coefficients transferred to the shared memory, being necessary a host-level synchronization before starting with the next gate. For these cases a simpler solution is possible by operating directly over the global memory, where all the coefficients are stored. Although access times to global memory will be worse, no advantage will be attained by copying partially the coefficients into shared memory, as they will not be reused at all.

Each thread is on charge of computing a minimum closed group, operating in-place. More formally, if the global thread grid identifier of a thread is $tid = nBlocks(blockId - 1) + threadId$ such a thread will be in charge of computing the closed group $G^i_{zero\_bit\_insert(tid,i)} = \{\alpha_{zero\_bit\_insert(t,i)}, \alpha_{zero\_bit\_insert(t,i)\oplus 2^i}\}$, as coefficients from the same group only differ on bit i. Note that the workload for each thread comprises not only the computation of its two coefficients, but also reading from global memory and writing back in the same positions.

Observe that there will be always coalescing if the qubit i, where gate is applied, is not smaller than the warp size. However, when this condition is not accomplished, the lack of coalescing will give rise to an effective loss of bandwidth. For these cases, it will be more efficient to make use of shared memory similarly to the coalescing-aware implementation.

7. Experimental results

This section analyzes the experimental results obtained after application of the techniques described previously. Simulations were executed in a GeForce® 8800GTX GPU NVIDIA platform and the CUDA 1.1 programming model. Mayor features of the platform and CUDA are shown in Tables 3 and 4.

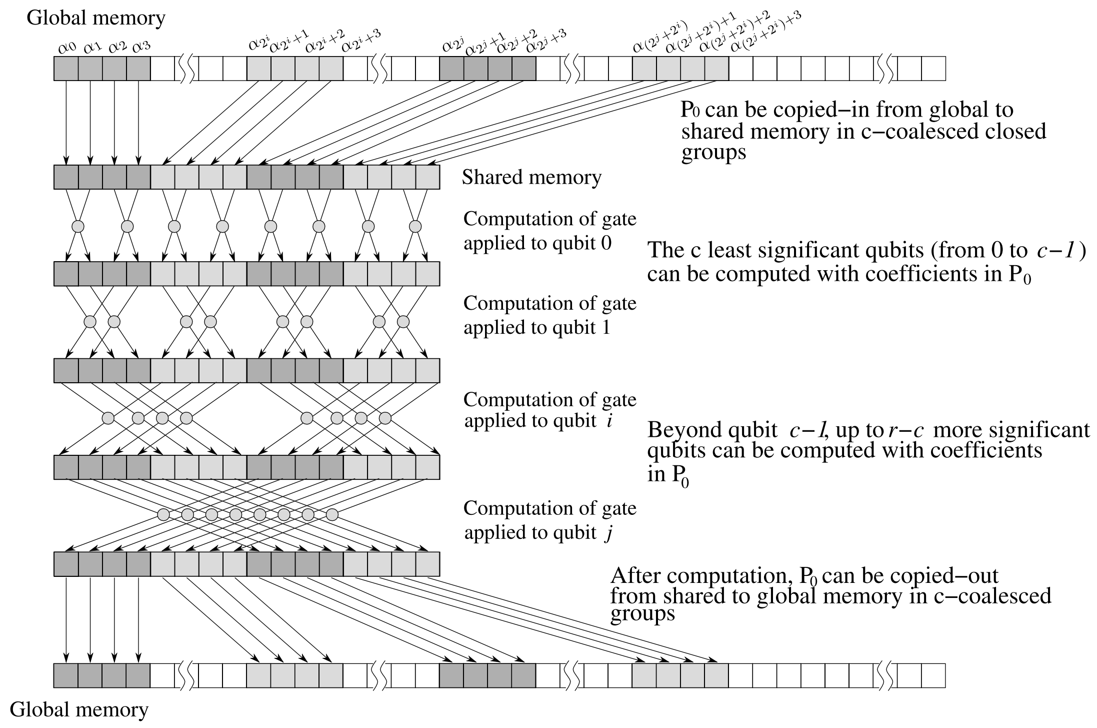

Global memory
α_0 α_1 α_2 α_3
α_{2^i} α_{2^i+1} α_{2^i+2} α_{2^i+3}
α_{2^j} α_{2^j+1} α_{2^j+2} α_{2^j+3}
$\alpha_{(2^j+2^i)}$ $\alpha_{(2^j+2^i)+1}$ $\alpha_{(2^j+2^i)+2}$ $\alpha_{(2^j+2^i)+3}$
P0 can be copied–in from global to shared memory in c–coalesced closed groups
Shared memory
Computation of gate applied to qubit 0
The c least significant qubits (from 0 to c−1) can be computed with coefficients in P0
Computation of gate applied to qubit 1
Computation of gate applied to qubit i
Beyond qubit c−1, up to r−c more significant qubits can be computed with coefficients in P0
Computation of gate applied to qubit j
After computation, P0 can be copied–out from shared to global memory in c–coalesced groups
Global memory

Fig. 15. Coalescing-aware strategy for a block. Consecutive threads transfer consecutive memory locations while copying-in/out. Each thread is in charge of computing a pair of coefficients operating in-place.

Table 3
Main physical features of the GeForce8800GTX platform.

Feature	Value
Multiprocessors in the GPU	16
Processors per multiprocessors	8
Clock frequency	1.35 GHz
Global memory	768 MB
Global memory latency	300+ cycles
Shared memory per multiprocessor	16 KB
Shared memory latency	4 cycles
32-bit registers per multiprocessor	8192

Table 4
Some features of CUDA 1.1 programming model related to the scheduling.

Feature	Value
Warp size	32 threads
Max. number of threads per block	512
Max. dimensionality of a grid	65 535 × 65 535 blocks
Max. dimensionality of a block	512 threads per dimension
Max. number of active blocks per multiprocessor	8
Max. number of active threads per multiprocessor	768

Several of the physical characteristics influence directly over simulation parameters (n, c and r). As each complex coefficient is represented by two 32-bit floats, the maximum number of qubits derives from the allocatable coefficients in global memory, $n = 26$ in this case. Besides, the shared memory limits the number of coefficients assigned to each block, that is, the cardinality of the coalesced planes P_k (parameter r). Note that cardinality has been expressed as a power of 2 and other thread variables are also placed in shared memory. This way the maximum number of coefficients that can be allocated in shared memory, results in 2^{10} ($r = 10$). Finally, parameter c, establishing the degree of coalescing is very much related to the scheduling of the threads into warps. So, though limited by r, an optimum is expected around the number of threads per warp. Table 5 summarizes the impact of the platform features.

The kernels were compiled with `nvcc` from NVIDIA. In order to obtain the best efficiency, configuration parameters (r, c) are fixed, defined as macros. Although a different binary is generated for each configuration, it helps the compiler to optimize the binary for each case. Otherwise several compiler optimizations would be prevented, and even more, the number of registers per thread would be higher, limiting the number of active threads at execution.

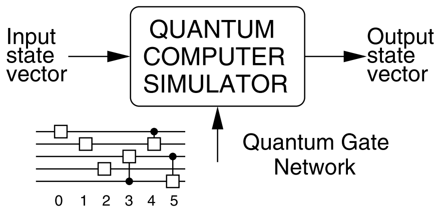


Fig. 16. Operation of our ideal quantum computer simulator.

An essential input for the simulator is the description of the network to be simulated (Fig. 16). It consists on a sequence of gates in the simulation order.

Experimentally two quantum transformations of interest have been addressed: a multiqubit transformation constituted by the Kronecker product of 1-qubit gates ($U^{\otimes n}$) and QFT. Within an experimental framework, the factors that influence the simulator performance are to be analyzed so that the best configuration can be selected for each particular instance.

For the first case, the base transformation has been the Hadamard gate ($U = H$), which results in the Walsh transformation. In this case the network of input gates is a sequence of gates H, each one applied to one qubit. In the case of QFT, as shown in Fig. 3(a), the gate network consist on a sequence with $n(n+1)/2$ gates. However it is possible to find out a more efficient equivalent configuration grouping the gates into steps, as shown in Fig. 3(b), existing n transformations Q_i. Notice that the computational complexity of Q_i is similar to a 1-qubit gate, as just one qubit is controlled by others less significant. If no other information provided, all implementations analyzed next are implemented based on the organization into steps.

7.1. Impact of the coalescing

The scheduling of threads into warps is reflected in the coalescing condition, that affects the copy-in/out stages of the simulation. The parameter c guarantees that transfers between global and shared memories are performed with a minimum coalescing of 2^c coefficients. Fig. 17 shows the influence of c on the execution time of the Walsh and QFT transformations of size n qubits applied over

Table 5
Impact of the platform features on the simulation, assuming two 32-bit floats per coefficient.

Platform feature	Simulation parameter	Max. value
Global memory size	Number of qubits: n	26 qubits ($n = 26$)[a]
Shared memory size	Coefficients per coalesced planes: 2^r	2^{10} coefficients ($r = 10$)[b]
Max. threads per block	Number of threads per block: 2^{r-1}	2^9 threads ($r = 10$)
Warp size	Coalescing degree in coefficients: 2^c	2^{r-1} ($c = r - 1$)

[a] 768 MB/(8 B/coeff.) = 96 Mcoeff./global $\geqslant 2^n \Rightarrow n = 26$, for in-place computation.
[b] 16 KB/(8 B/coeff.) = 2^r + extra space for variables $\Rightarrow r = 10$.

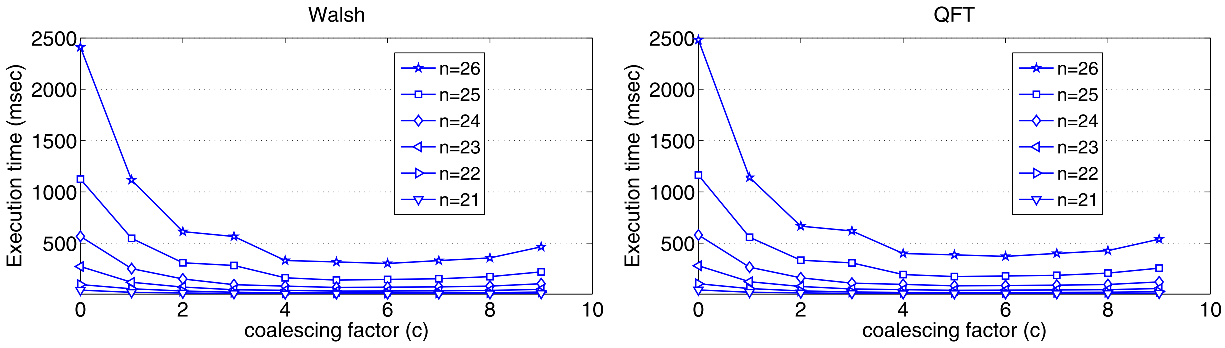


Fig. 17. Impact of the coalescing factor on the execution time.

an n-qubit register. The data were obtained for the maximum possible cardinality $r = 10$, and a variable number of qubits in the register.

Experimental results do confirm this. Observe that the execution time decreases as the coalescing approaches the warp size. A minimum is reached from $c = 4$, with a slight increment for greater values of c. This increment is due to the increase in the total number of stages with the increment of c (Eq. (10)). For a certain cardinality, a smaller c translates into less gates per stage and then, a larger number of copy-in/out and host synchronization operations. If it is less than 4, computing time worsens drastically due to the serialization of the accesses to global memory caused by the lack of coalescing. If it is above 7, execution time is slightly incremented due to the overhead produced by the opening and closing of an increasing number of stages. Thus, the best performance is attained for c values in the range from 4 to 7.

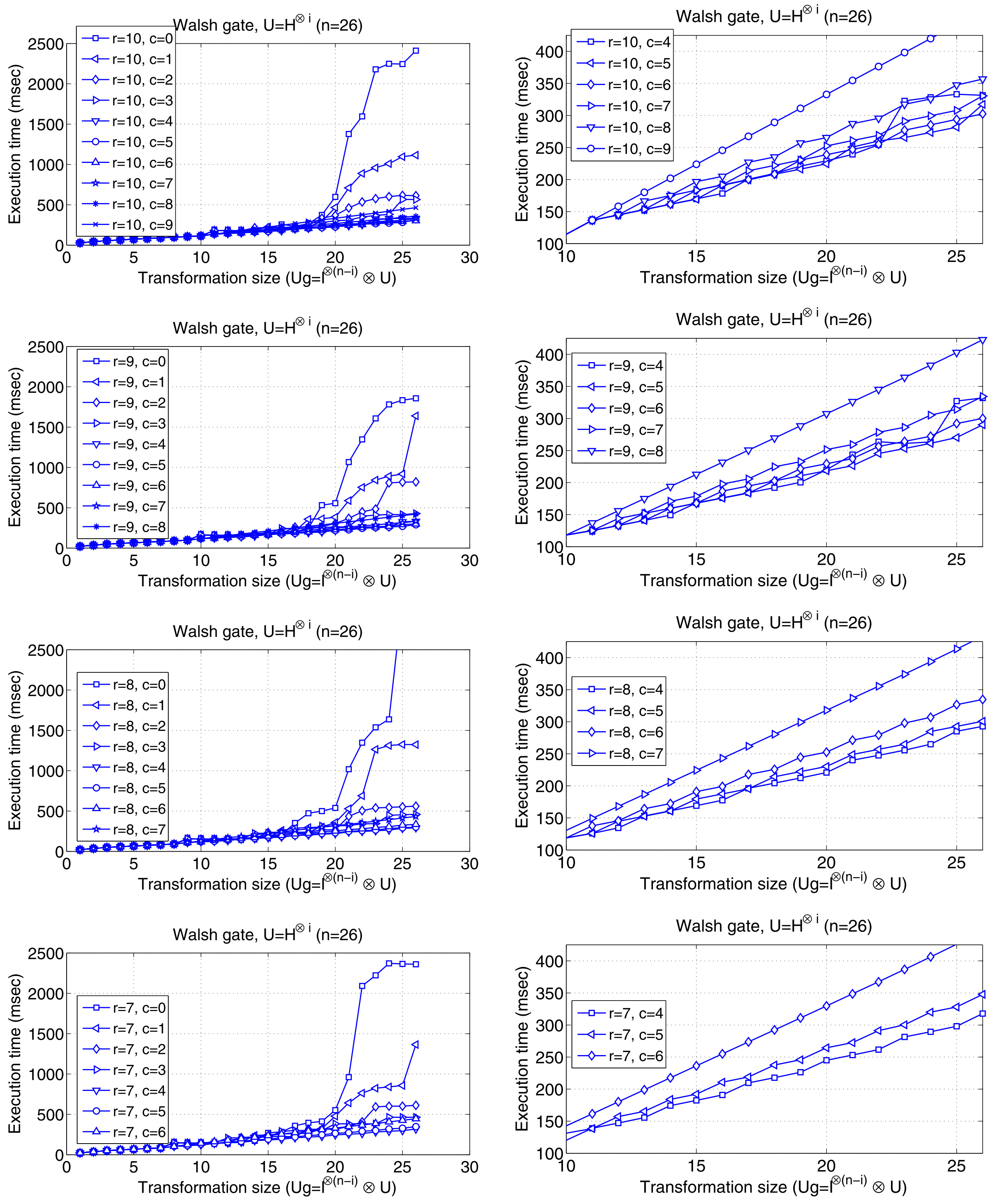


Fig. 18. Results after applying the Hadamard gate to a variable number of qubits of a 26-qubit register for different values of coalescing and cardinality.

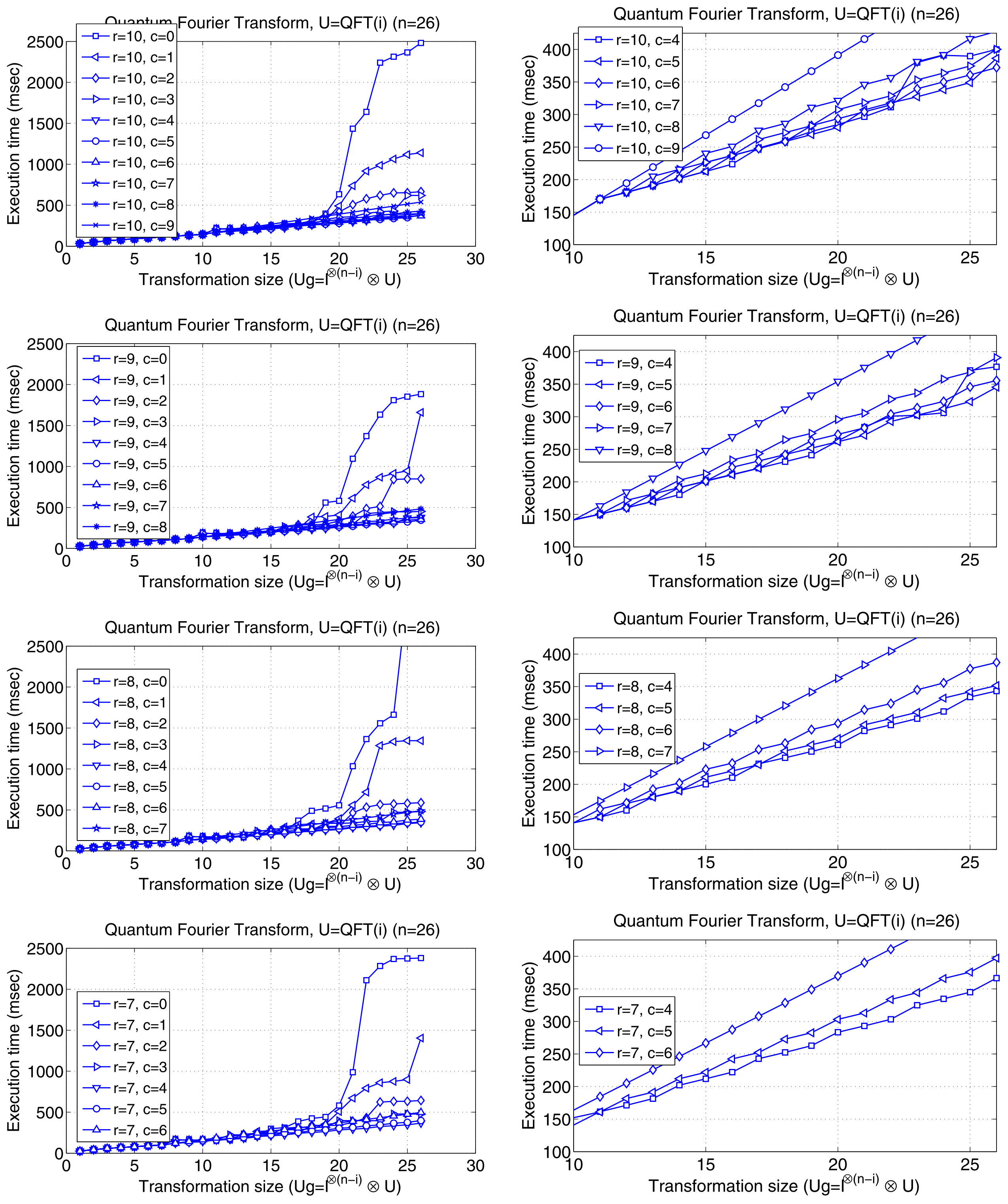


Fig. 19. Results after applying the QFT to a variable number of qubits of a 26-qubit register for different values of coalescing and cardinality.

7.2. Impact of the cardinality

Transformations applied over a variable number of consecutive qubits (from 0 to $i-1$) of a register with $n = 26$ qubits, are shown in Figs. 18 and 19 for a wide range of simulation configurations. Plots on the right show a zoom of an area of particular interest. Note that curves are modulated by the partition of the gate network into simulation stages (Eq. (10)). This way, there is only one stage for the r first points, resulting in a linear growth of the execution time for all the configurations. Different values of r give place to steps each $r - c$ qubits, corresponding to the initialization of each new stage. This overhead is to be amortized with the next $r - c$ gates. A particular case is when $c = r - 1$, for which stages beyond gate r have only one gate. Here, copy-in/out operations cause the main overhead, resulting in a linear graphic.

The cardinality parameter r not only determines the number of stages and gates by stage, but also the size of the coalesced planes and thus, the number of blocks and threads. For an i-qubit trans-

Table 6
Active blocks per multiprocessor are restricted by platform scheduling features.

Limitation	# active blocks subject to
Number of active blocks/multiprocessor	$Nb \leqslant 8$
Number of active threads/multiprocessor	$Nb \cdot 2^{r-1} \leqslant 768$
Number of active warps/multiprocessor	$Nb \cdot 2^{r-1}/32 \leqslant 24$
Number of threads/block	$2^{r-1} \leqslant 512$
Shared memory/multiprocessor	$Nb \cdot 2^{r} \leqslant (2^{11}$-extra space for variables)
Total # of 32-bit registers/multiprocessor	$Nb \cdot 2^{r-1} \cdot$ registers/kernel $\leqslant 8192$

Table 7
Occupancy achieved by different configurations.

r	t/b	shmem/b	# a.b.	# a.t.	Occupancy	Limiting factors
10	**512**	**8 KB** + *vars.*	1	512	2/3	shmem/b, t/b
9	256	**4 KB** + *vars.*	3	**768**	1	shmem/b
8	128	2 KB + *vars.*	6	**768**	1	–
7	64	1 KB + *vars.*	**8**	512	2/3	# a.b.
6	32	512 B + *vars.*	**8**	256	1/3	# a.b.

t/b = threads per block; shmem/b = shared memory per block; a.b. = active blocks per multiprocessor; a.t. = active threads per multiprocessor; *vars.* = extra space for variables.

formation, the number of CUDA blocks is 2^{i-r}, and the number of threads per block is 2^{r-1}. After these, it seems that it would be optimum to consider the maximum allowable size, $r = 10$ (1 K coefficients in shared memory).

However, another factor to take into account is the *occupancy*, as several active blocks may coexist in the same multiprocessor if there are enough registers and shared memory available. Occupancy is the ratio between the total number of active threads and the maximum allowable (see Table 4). Thus, if $r = 10$, then $2^{r-1} = 512$ threads/block, and it is not possible to have more active blocks in the same multiprocessor, as there is not enough space in shared to allocate more coefficients. When $r = 9$, it is $2^{r-1} = 256$ threads/block, and the number of coefficients per block occupies 4 KB. This frees space in shared that can be used by another blocks, rising the occupancy. Table 6 shows the restrictions imposed over the number of active blocks (Nb), whereas Table 7 provides occupancy values for different values of r, highlighting which is the limiting factor for each of them. A maximum occupancy is reached for $r = 9$ and $r = 8$ configurations. Another limiting factor is the number of registers that each thread demands, determined at compile time. All our simulator kernel codes are tuned to fulfill this limit.

In a strict sense, if there are more threads than processors per multiprocessor, an occupancy below the unit does not involve necessarily a lower effective parallelism. Nevertheless, a smaller occupancy means fewer opportunities for hiding latency, which worsens the effective memory bandwidth.

The factors determining the computation time for a certain parametrization of the simulation (n, r and c) result to be the coalescing, the number of blocks and threads, the occupancy and the shared memory size. Observe that the behavior of the curves in different experiments is quite regular. Almost in all the cases, the optimum value is reached for $r = 9$ when c is between 4 and 7.

7.3. Bandwidth

Fig. 20 shows execution time when just one isolated gate located in qubit i of a 26-qubit register is simulated, with a maximum entropy initial state. This experiment illustrates the fact that not using the shared memory or not reusing data results in a poor performance. In general, when there is no data reuse it is better to work directly in global memory (gate-by-gate implementation). However, a lack of coalescing is even worse, as it happens for the first qubits. Nevertheless if there is more than just one gate per

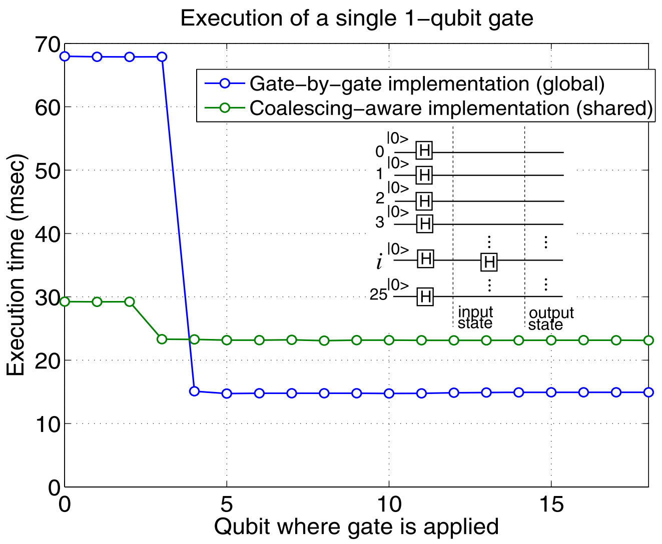


Fig. 20. A single 1-qubit gate does not amortize the use of the shared memory.

stage, it has been demonstrated that the coalescing-aware implementation is advantageous, since it amortizes existing overheads.

There are two sources of overhead: transference between memory spaces and the index computation. In order to quantify these overheads a synthetic identity transformation has been coded, involving only copy-in/out and computations of the indexes mapping memory spaces, without the computational payload. Assuming that the execution time of the transformation under test (T_U) includes the same overheads as the reference identity (T_I), an efficiency measure representing the payload over the total time would be: $\eta = \frac{T_U - T_{Iref}}{T_U}$.

A performance model can be proposed for the coalescing-aware implementation, considering several components in the execution time: computational payload ($Tcomp$), index computations ($Tind$), copy-in/out and host synchronization efforts (Tcp) and warm-up time associated to the kernel launching (I): $T_U = Tcomp + Tind + Tcp + I$.

Concerning to the execution time of a block, the payload and indices computation times will be proportional to the number of coefficients to be processed per such a block (2^r) and the number of gates: $Tcomp = K_1 2^r i$, $Tind = K_2 2^r i$. The overhead imposed by copying-in/out will be proportional to the number both of stages ($nStages$ after Eq. (10)) and coefficients to transfer. If the overhead of the first stage is considered independently, it will be $Tcp = K_3 2^r(nStages - 1)$. Thus, the term I will include the overhead associated to this first stage as well as those associated to kernel launchings. This way it will be:

$$\eta = \frac{K_1 2^r i}{(K_1 + K_2)2^r i + K_3 2^r(nstg - 1) + I},$$

$$\text{with } nstg = \begin{cases} 1, & \text{if } i \leqslant r, \\ 1 + \lceil \frac{i-r}{r-c} \rceil, & \text{if } i > r. \end{cases} \tag{11}$$

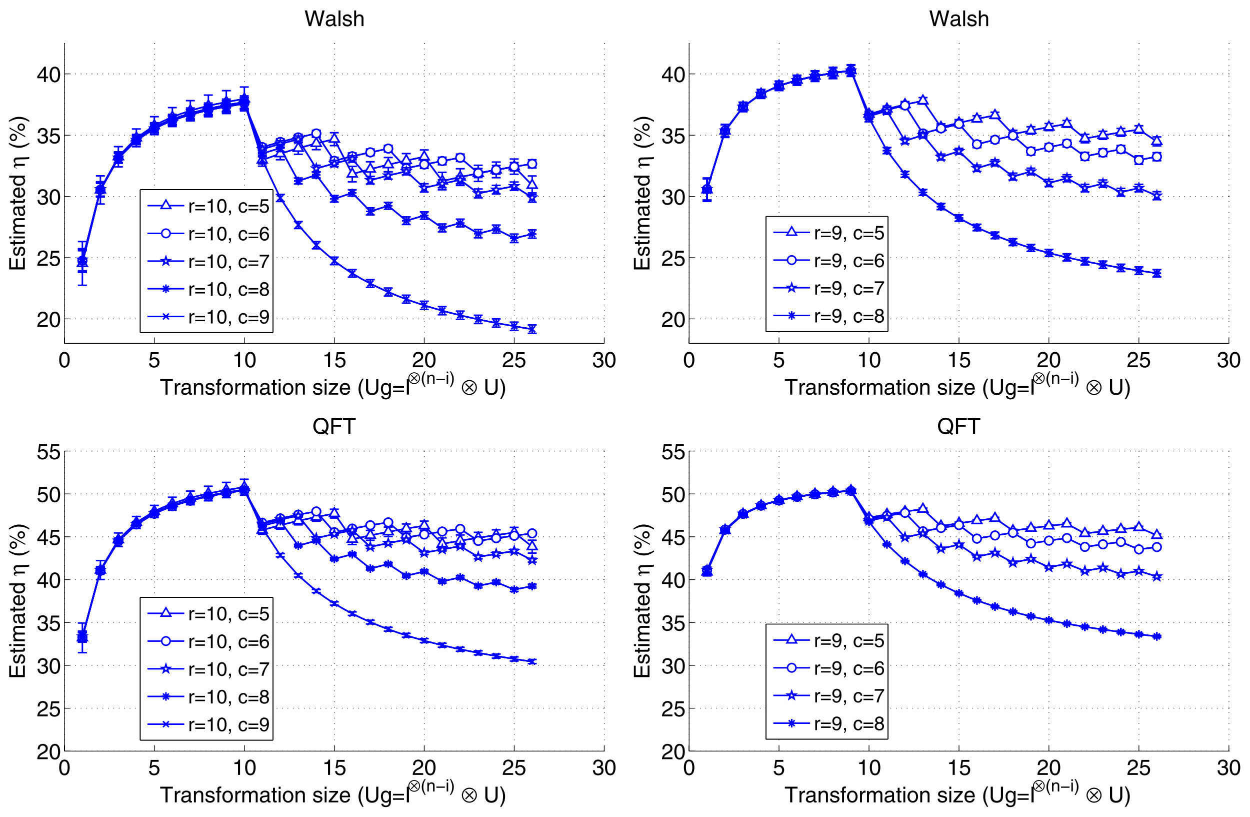


Fig. 21. Estimation error plots of the model, for Walsh and QFT applied to a variable number of qubits with $n = 26$.

Table 8
Parameter estimation for the execution model. Components of the execution time are determined for a transformation size equal to the register size.

n	r	c	c.i. radius	Relative error	Computational payload	Index calculation	copy-in/out
Walsh:							
26	8	5	28%	1.2%	35%	45%	20%
26	8	6	28%	1.2%	31%	41%	28%
26	8	7	28%	1.4%	24%	32%	44%
26	9	5	33%	1.4%	36%	47%	17%
26	9	6	32%	1.4%	34%	46%	20%
26	9	7	31%	1.4%	31%	41%	28%
26	9	8	31%	1.5%	25%	32%	43%
26	10	5	55%	3.1%	32%	45%	23%
26	10	6	28%	1.5%	33%	49%	18%
26	10	7	28%	1.6%	30%	45%	24%
26	10	8	27%	1.6%	28%	40%	32%
26	10	9	30%	2.0%	20%	28%	52%
QFT:							
26	8	5	12%	0.5%	43%	40%	17%
26	8	6	12%	0.5%	40%	36%	24%
26	8	7	11%	0.5%	33%	30%	37%
26	9	5	15%	0.5%	45%	42%	13%
26	9	6	12%	0.4%	44%	41%	15%
26	9	7	10%	0.4%	40%	38%	22%
26	9	8	11%	0.4%	33%	31%	36%
26	10	5	42%	2.2%	44%	37%	19%
26	10	6	12%	0.6%	45%	40%	15%
26	10	7	13%	0.7%	42%	37%	21%
26	10	8	13%	0.7%	39%	34%	27%
26	10	9	15%	0.9%	31%	26%	43%

In order to validate this model, a nonlinear regression method is applied. Estimation error plots for predicted values are shown in Fig. 21. Observe that for parameters implying less gates per stage (lower $r - c$), the overhead increases, as less advantage is taken of the operations that produce the overhead. It is similar when a new stage begins. It produces a small local decrease in the curves, as associated overhead will not be compensated until there will not be enough gates. Table 8 shows results obtained from prediction, that cast low relative errors, in most cases below the 2%. Fig. 22 displays experimental results confirming the different estimated time components. These graphics not only show times for Walsh, QFT, and the reference identity, but also for another synthetic transformation which only performs copy-in/out for the stages, without computation of either indices or coefficients.

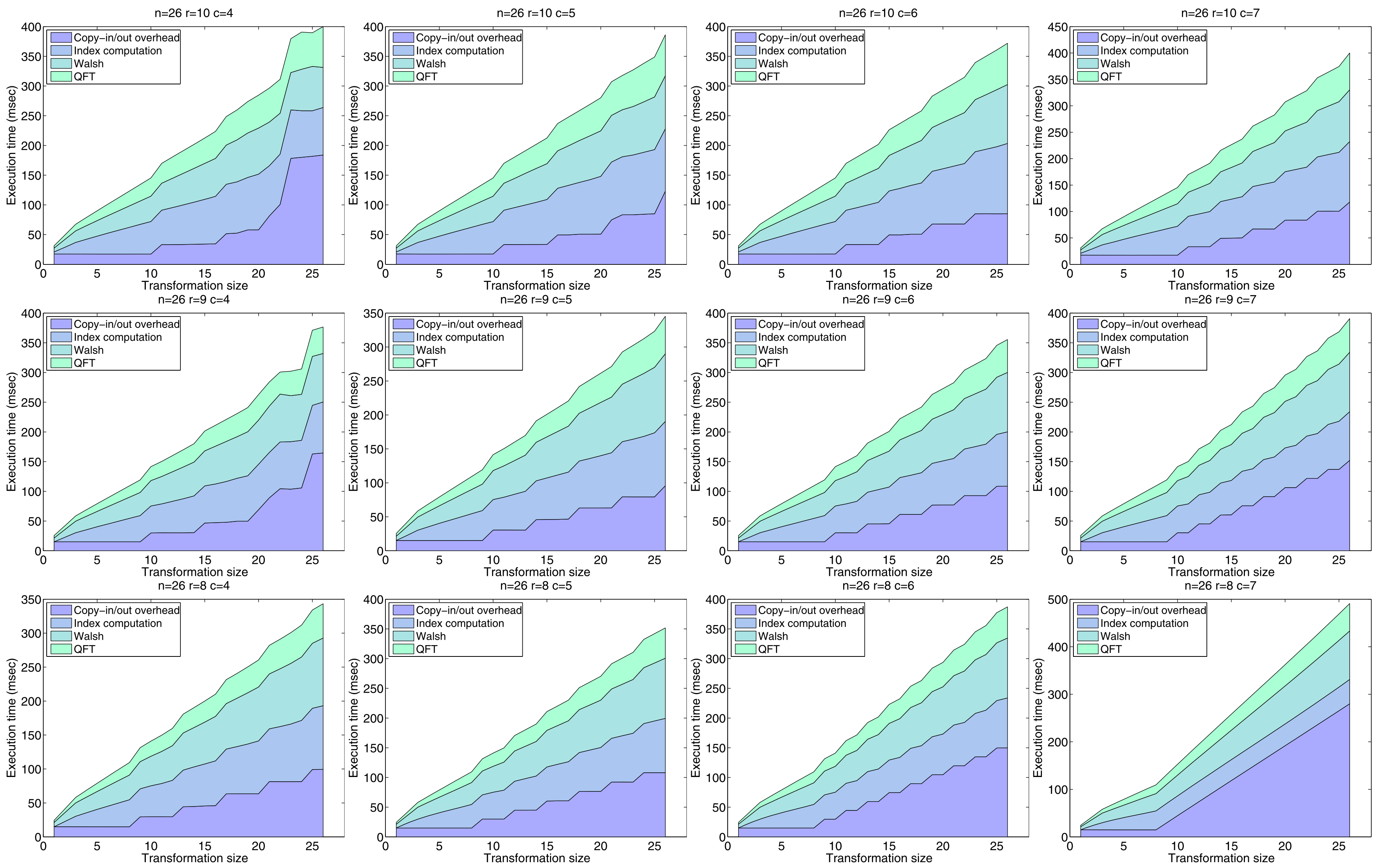


Fig. 22. Components of the measured execution time for different simulation parameters when Walsh gate and QFT applied to a variable number of qubits.

Table 9
Execution time of Walsh and QFT experiments for different implementations.

n	CUDA programming model on GPU						Sequential code on CPU		Speed-up
	Gate-by-gate	Coalescing-aware					Libquantum	Optimized	t_{CPU}/t_{GPU}
	t (msec)	r	c	nstg	last	t (msec)	t (msec)	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

These results bring out that the strategies focused on the exploitation of the shared memory involve unavoidable overheads. This fact translates into a limitation of the higher possible r by the shared memory size and the occupancy.

7.4. Scalability

This subsection analyzes how the different strategies under study scale with the problem size. For comparative purposes, Table 9 shows the execution time (in msec) of the parallel strategies, gate-by-gate and coalescing-aware, and two sequential implementations on a CPU. For the coalescing-aware one, r and c correspond with the fastest configuration. Also, the number of generated simulation stages, according to Eq. (10), is displayed in column *nstg*. This corresponds to the number of code kernels to be invoked. Quotients in column *last* represent the number of gates remaining in the last stage with respect to the number of gates by stages $(r - c)$. This last fraction provides a measure of the boundary effect associated to the last stage. A value less than one involves a worse data reuse than all previous stages.

Sequential executions were conducted on a 2 GB RAM PC with an Intel Core2 6400 @ 2.13 GHz. Results are obtained with the *libquantum* library [4] and also with an optimized and simplified sequential version, developed by the authors based on the *libquantum*'s source code. This version minimizes unnecessary overheads, retaining the essential functionality required for testing purposes, which allows a fairy comparison with the GPU executions.

In the case of Walsh transform, where there are as many gates as qubits, the expected execution time for sequential codes grows after $\mathcal{O}(n2^n)$ when applied to a n-qubit register. In the case of QFT, where the number of gates is $n(n+1)/2$, the execution time is higher as shown for *libquantum*. Nevertheless by grouping QFT gates into steps (see Fig. 3) the order $\mathcal{O}(n2^n)$ is kept. This last approach has been followed in the optimized sequential code as well as in all the implementations on GPU.

An important fact derived from Table 9 is that coalescing-aware simulation always presents a better execution time than the gate-by-gate one. This points out that it is mandatory to exploit the GPU memory hierarchy when high performance is pursued. Observe that, specially for high number of qubits, n, the smallest execution time is attained for the parameters r and c fitting the device features. In this way, the optimum r provides the maximum GPU occupancy, limited by the shared memory size. The parameter c, closely related with the data coalescing, is always 4 or 5, that is, a half warp (16 threads) or a warp (32 threads).

The rightmost column, *speed-up*, is the quotient between execution time of the CPU optimized sequential version and the GPU coalescing-aware implementation. QFT experiments provide a better speed-up when compared with Walsh due to the higher computational workload of the QFT, that results on relatively less significant parallelization overhead. Note that parallel implementations on the GPU exhibit a good scalability as speed-up does not degrade when the problem size increases.

Although it is difficult to compare heterogeneous platforms, the proposed coalescing-aware strategy is near to reach an speed-up of 100 in the best cases, when compared to the CPU. This is a considerable figure, taking into account that the GPU device has 128 stream processors.

8. Conclusions

Approaches to the simulation of an ideal quantum computer using the CUDA programming model on NVIDIA GPUs have been proposed and analyzed in this paper. It can be pointed out that the explicit exploitation of the fast on-chip memory via latency hiding, is an essential issue when high efficiency is pursued. Additionally, this work develops an analysis framework that allows to select simulation parameters concerning some major platform details. It takes into account important features like the memory reference locality, the size of the on-chip memory and the processor occupancy. From this analysis, configurations for optimum simulations are experimentally obtained and evaluated. The results show that the proposed strategies are able to achieve high speedups on these kind of commodity multiprocessors. On a typical GPU of 128 stream processors a 26-qubit register can be simulated up to one hundred times faster than the sequential code in a coeval conventional monoprocessor platform.

References

[1] M. Aminian, M. Saeedi, M. Zamani, M. Sedighi, FPGA-based circuit model emulation of quantum algorithms, in: IEEE Computer Society Annual Symposium on VLSI, 2008, pp. 399–404.

[2] G. Arnold, T. Lippert, N. Pomplun, M. Richter, Large scale simulation of ideal quantum computers on SMP-clusters, in: Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, in: John von Neumann Institute for Computing Series (NIC), vol. 33, Central Institute for Applied Mathematics, Jülich, Germany, ISBN 3-00-017352-8, 2005, pp. 447–454; DBLP, http://dblp.uni-trier.de.

[3] A. Barenco, C. Bennett, R. Cleve, D. DiVicenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, H. Weinfurter, Elementary gates for quantum computation, Phys. Rev. A 52 (5) (1995) 3457–3467.

[4] B. Butscher, H. Weimer, Libquantum library, at http://www.libquantum.de.

[5] D. Deutsch, Quantum computational networks, Proceedings of Royal Society of London Series A 425 (1989) 73–90.

[6] D. Deutsch, R. Jozsa, Rapid solution of problems by quantum computation, Proceedings of Royal Society of London Series A 439 (1992) 553–558.

[7] A. El Zein, E. Mccreath, A. Rendell, A. Smola, Performance evaluation of the NVIDIA GeForce 8800 GTX GPU for machine learning, in: Lecture Notes in Computer Science, vol. 5101, 2008, pp. 466–475.

[8] M. Fujishima, FPGA-based high-speed emulator of quantum computing, in: IEEE Int'l Conference on Computer Design, 2004.

[9] I. Glendinning, B. Ömer, Parallelization of the QC-lib quantum computer simulator library, in: Lecture Notes in Computer Science, vol. 3019, 2004, pp. 461–468.

[10] L. Grover, A fast quantum mechanical algorithm for database search, in: Annual ACM Symposium on the Theory of Computation, 1996, pp. 212–219.

[11] P. Ha, P. Tsigas, O. Anshus, The synchronization power of coalesced memory accesses, in: Lecture Notes in Computer Science, vol. 5218, 2008 pp. 320–334.

[12] A. Khalid, Z. Zilic, K. Radecka, FPGA emulation of quantum circuits, in: IEEE Int'l Conference on Field-Programming Technology, 2003.

[13] L.P.M. Udrescu, M. Vladutiu, Using HDLs for describing quantum circuits: A framework for efficient quantum algorithm simulation, in: Computing Frontiers Conference, 2004.

[14] M. Nielsen, I. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2004.

[15] J. Niwa, K. Matsumoto, H. Imai, General-purpose parallel simulator for quantum computing, Phys. Rev. A 66 (6) (Dec. 2002) 062317–062328.

[16] NVIDIA CUDA: Programming guide, and SDK, at http://www.nvidia.com/cuda.

[17] K. Obenland, A. Despain, A parallel quantum computer simulator, at http://arxiv.org/abs/quant-ph/9804039, 1998.

[18] B. Ömer, QCL: A programming language for quantum computers, at http://tph.tuwien.ac.at/~oemer/qcl.html.

[19] Quantum computer simulators, at http://www.quantiki.org/wiki/index.php/List_of_QC_simulators.

[20] K.D. Raedt, K. Michielsen, H.D. Raedt, B. Trieu, M.G. Arnold, Massively parallel quantum computer simulator, Computer Physics Communications 176 (2007) 121–136.

[21] P. Shor, Algorithms for quantum computation: Discrete logarithm and factoring, in: Symposium on Foundations of Computer Science, 1994, pp. 124–134.

[22] SPEC 2006: CPU-intensive benchmark suite, at http://www.spec.org.