Politecnico di Milano
**DRESD** Project
nomeprogetto



# QCADESIGNER FOR CUDA - COHERENCE VECTOR ENGINE

2010

# Contents

**Abstract**

The development of tools capable of simulating the behaviour of QCA based circuits is a fundamental and necessary step required for this technology to make progresses and to allow the fabrication of real world devices. We focus on the work of the MiNa Group from University of Columbia, and in particular on their QCADesigner simulation IDE/Engine. At their current development stage, even a simple circuit requires many hours of simulation in order to characterize its dynamic. In this work we focus on the bottlencks of their implementation and develop a parallelized solution based on nVidia CUDA technology, enabling a speedup of the performances (objective: execution time) proportional to the number of symmetric multiprocessors available on the system employed for simulation (with respect to the same circuit executed on a serial machine).

# Chapter 1

# Project Introduction

Computer technology has made incredibly progresses since the first general purpose computer was created, and not until nowadays, the annual performance growth has been steadily growing by 35-60% per year. It has been only recently that this growth has reduced, mainly because the reach of the limits in the miniaturization of silicon-based transistors (and the associated heat disspiation rates and limits into clock frequency improvements), intrinic limits into ILP exploitation and almost unchanged memory latencies.

In order to overcome some of these problems, in particular those related to the physical limits of silicon-based technologies, a number of approaches are now under development. DNA, membrane and quantum computing are the major areas of reasearch, focusing not only on theoretical, complexity-related aspects of the subject, but also on the physical machineries exploitable to realize these kinds of computations.

In our work we focus on a technology called *Quantum Dot Cellular Automata cells*, (or, simply, QCA). At the heart of this technology there are quantum mechanics and nanotechnology. The former describes the behaviour of the charges flowing through these cells (*ie*, the dynamics of this system), while the latter allows for the fabrication of small enough structures capable of maintaining charges in proper position for the required time.

Even though it has even been demostrated (solely from a mathematical point of view) the the possibility of using QCA cells for building quantum computers - something quite not at hand yet - they can nonetheless be seen as powerful alternatives to silicon-based devices to represent and transmit data. Depending on the fabrication technique employed, in fact, QCAs are theoretically able to switch at orders of magnitude around the THz, to be packaged at very high density levels and to dissipate very small amounts of power (compared to the current silicon-based technology). This, along with the existance of demo systems already proven working, justifies their current and future development as an attractive way of improving nowadays physical computing bottlenecks.

One of the institutions interested and involved into the development of this technology, among the others, is the University of Columbia, and in particular its Microsystems and Nanotechnology Group (MiNa Group). The foundation of our work is their implementation of a simulator (both logical and physical) of QCA based cicuits, QCADesigner.

Our focus is set on the improvement of the performance of this application (objective: execution time) to the point that is feasible to simulate sequential, large sized circuits.

# Chapter 2

# QCA Overview

Quantum Dot Cellular Automata (QCA) are a quantum extension to the classical notion of Cellular Automata.

In computability theory, we identify a Cellular Automata (CA) as a Turing-complete abstract machine model consisting of a grid of *cells* each of which can be in any of a finite number of *states*. For every cell in the grid, moreover, we define its *neighborhood*, which is the set of "near enough" cells. The *evolution* of this automaton is defined by the transition of the state of the cells, which is a function of both the status of the cell and the status of its neighborhood.

What makes a QCA different from a classical CA is that its physical implementation is regulated by the quantum laws of physics instead of the classical ones. Let's take a closer look to a QCA. As you can see in Figure 2.1 a QCA is made of three main components: electron wells, crystalline substrate and electrons. Here we depict the charge of the electrons on the wells/dots by blurred blue clouds instead of the more classical representation of small, crisply defined points. This is accurate because the Quantum Mechanics says that at this near atomic scale, charge (on electrons and protons) does not come in sharp little balls, but instead is correctly representable only in a probabilistic way via the wave function (Schrödinger equation).

Charges of the same type (positive or negative) still repel one another so our two clouds of charge will move as far from the another as possible by moving to opposite corners of the QCA (this allows for the largest distance to be set between the clouds). In Figure 2.2 is represented the typical evolution of a QCA cell. It is quite easy to generalize the represented mechanism to a situation where an arbitrarily large number of cells are put adjacently in order to form, for example, a so called *quantum wire*. In such a configuration, whenever a cell switches from a state to the opposite, so does its neighbour and the neighbour of the neighbour... and so on until the end of the wire. This mechanism allows for information to be *transmitted*.

But what is the phenomenon that allows a charge to move precisely from a well to the other? This is called "tunnel effect" and it is a well known phenomenon in quantum physics stating that, under certain hypothesis, waves (in our case: the charge cloud) are enabled to move through thin layers of obstructing materials (in our case: the walls of the wells). This is the main distinguishing reason why these cells are not simply cellular automata but *quantum* ones. This must not lead the reader to think that the cells, just *per se*, allow for quantum computing to take place. In fact, these devices, as for how have been described, always behave in a *deterministic* manner, being polarized in one *or* the other configuration. Quantum computing by means of QCA cells exploitation has been demonstrated feasible from a mathemathical point
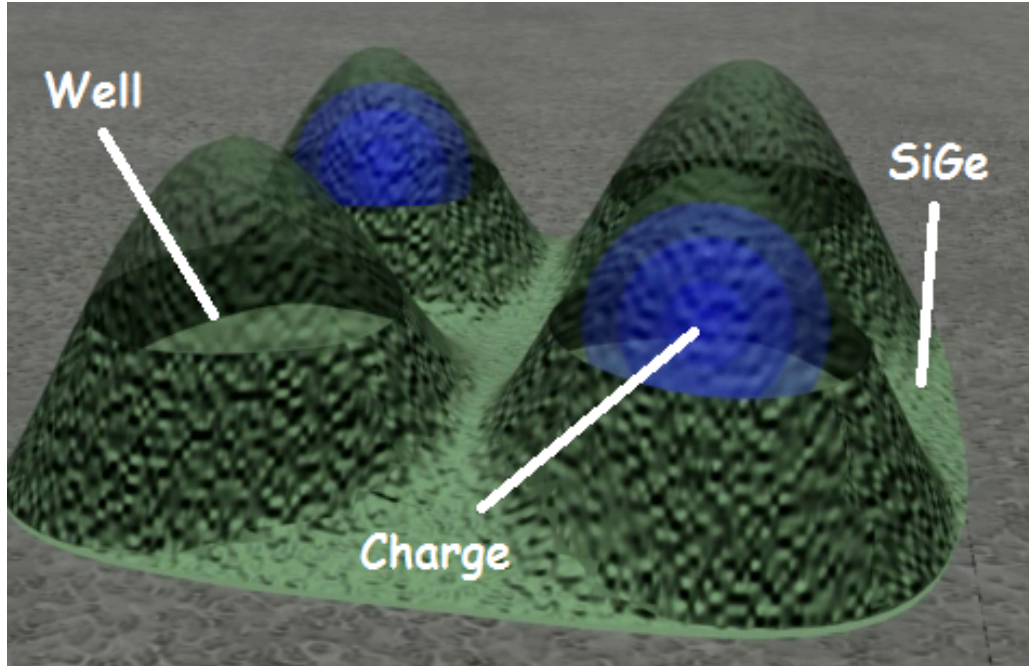
Figure 2.1: A single Quantum Dot Cellular Automata (QCA) cell. In green the crystalline (SiGe - Silicon Germanium) substrate, where are clearly visible, in squared position, the four wells meant for accomodating that many electrons. Even though different configurations of the dots are possible, this is by far the most commonly studied structure. The blue cloud represents the amplitude of probability of finding an electron in that position.

of view, but no real world implementation, to date, have publicily been demonstrated.

The only (from a theorethical point of view) remaining feature that is required to enable the *controlled/programmable* transmission of data troughut a circuit of adjacent QCA cells is the concept of "clock". Clocks are areas of conductive material under the circuit's lattice, modulating the electron tunneling barriers in the QCA cells above it. This way it possible to "strengthen" the barriers disallowing charges to flow (even under those conditions where they would naturally do it) or, as opposite, to "weaken" them to favour movements and changing of states.

An example of a QCA cells based circuit it is reported in Figure 2.3. The conjunction of adjacent electrical fields of the three input drivers results into the polarization of the central cell as *the opposite* of the majority of them, thus inducing the rightmost cell to polarize *like* them (thus the name: Majority Gate). By means of the Majority Gate we can build the AND and the AND ports and, with a slightly more complex structure, the NOT one. Thus, any digital circuit can be reimplemented using this technolgy.
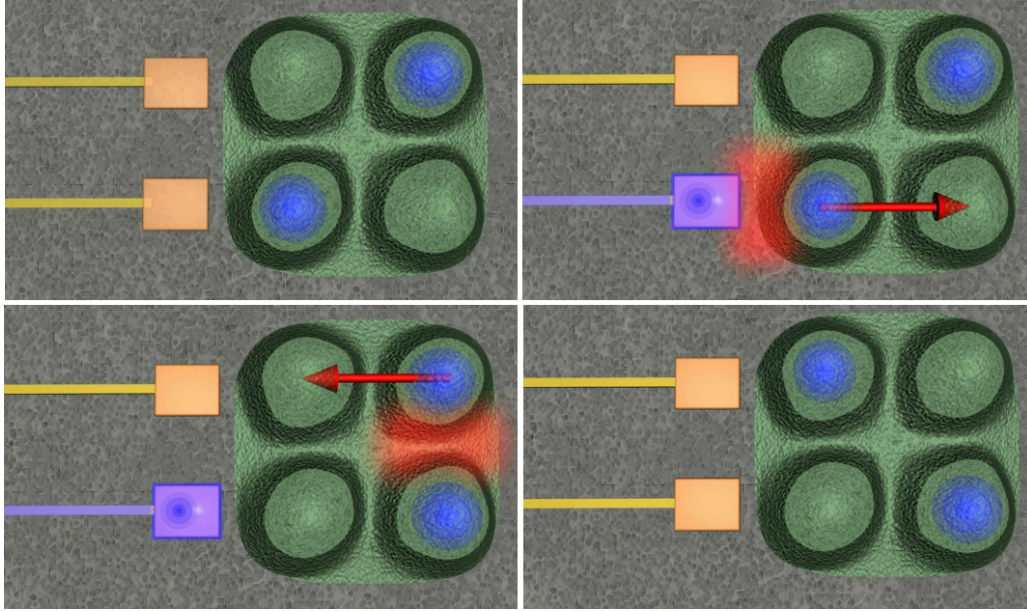
Figure 2.2: The evolution of the state of a QCA cell. From left to right, from top to bottom: the lower of the two metallic wires (on the left in every image) is polarized, inducing (second image) the lower left electron to flow to the rightmost lower side of the QCA (and precisely inside the corresponding well). This, in turn, induces the upper right electron to move farthest from it, inside the upper left well (third and fourth image).
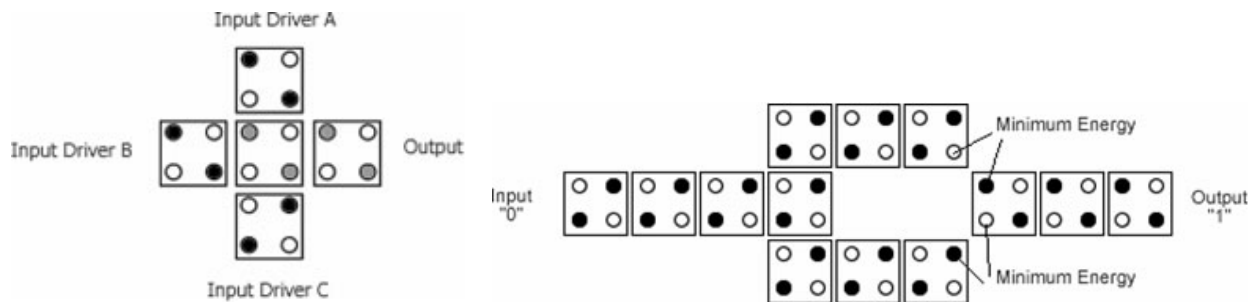


Figure 2.3: The majority gate drives the output cell's state to be equal to that of the majority of the inputs. The not port exploit a different angle between cells to obtain the negation of the input.

# Chapter 3

# CUDA Overview

CUDA is a software layer that allow programmers to exploit the capability of nVidia GPUs as general purpose processors.

Dealing with a video card in this way requires approaching a completely new programming style and acquiring some knowledge about the basic nVidia GPUs architectures, even though all the internal details are masked by the framework.

CUDA exposes the GPU as a "Parallel Co-Processor" that can be used by the CPU to speed-up the computations. More in the details, the CPU - Host - can take advantage of the high amount of parallel threads executable by the GPU - Device - to accelerate parts of a program that are especially well suited for exploiting TLP. According to this approach, the CPU must directly manage the program execution settings on the GPU, provide the data for the computation to the device and collect the outputs when it is done.

One of the most important features of CUDA is that it abstracts away all the physical details of the supported GPUs and always shows to the programmer the very same logical organization (Figure 3.1). These GPUs can be considered a MIMD array of SIMD processors , called MultiProcessors. Each MultiProcessor is composed of 3 elements: a fixed number of cores, an instruction unit and a private memory space. All the MultiProcessors share a pubblic memory space referred to as Device Memory in order to distinguish it from the CPU memory space - Host Memory - that is not directly accessed by the GPU. Due to the property of abstraction mentioned before, the only difference between families of nVidia products is in the amount of memory, the number of MultiProcessor and the nature of the cores.

## 3.1 Multiprocessors

Even if the architectural details of the CUDA capable GPU can significantly differ from a product to another, some common guidelines can be identified.

As stated before the MP is a SIMD component: all the SPs belonging to the same MP execute the same instruction on different data. This structure, obviously deigned for graphical purpouses, make the GPUs also particulary effective in addressing problems that can exploit a huge amount of TLP.

Form a philosophical point of view, the MPs of these GPUs are designed to be as simple and fast as possible. In order to keep low the complexity, there are neither branch predictors, nor mechanisms to rollback incorrect results. Even if this seems a serious limitation to the perfor-
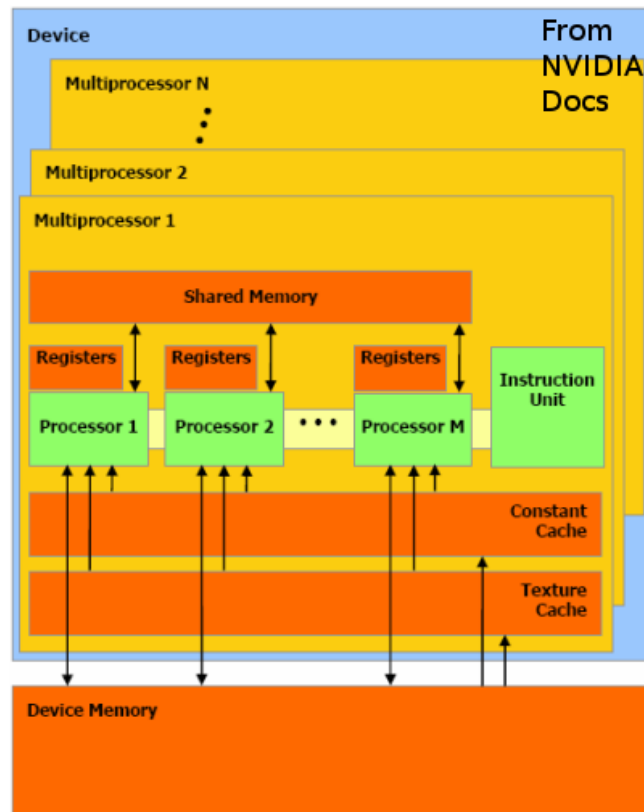
Figure 3.1: Logical Organization of all the CUDA capable devices. The programmer does not have to take care of the physical organization of the GPU that will actually execute the program.

mance, it allow the cores of the MP to be completely focused on the arithmetic intensity. As results of this choice, the SPs of the most recent nVidia products are able to perform a double precision MAD (or MUL or ADD) per clock cycle.

## 3.2 Memory Hierarchy

The CUDA memory hierarchy consists of several elements optimized for different memory usages.
The Device Memory is the main memory space of the GPU. It can be accessed by both the CPU and the GPU using different policies, usually with a latency of some hundreds of clock cycles. In order improve the performance and the usability, it is logically partitioned into three logic component, depending on the access method: the Global Memory, that follow the common 32-,64-, or 128-byte memory transactions paradigm; Texture Memory, accessed by texture fetching; Constant Memory, managed by special operations.
The Global Memory is the most frequently employed memory space. Usually it is used by the CPU to load the data for the computations into the device and by the GPU to provide the results. Furthermore the Global Memory is the only space completely shared among all the SPs of the device: for this reason it is also exploited for comunication among SPs belonging

to different MPs. The Texture Memory can be written by the CPU using the CUDA API and readed by the GPU via texture fetching. No GPU texture write mechanism is provided. The Constant memory is a small special memory space that can be used for allocation of variables frequently readed. These variables must be allocated by the CPU before the execution on GPU, that can access them solely in read-only mode.

Due to the high latency of the Device Memory, each MP is provided with a private low latency memory space, logically divided into:

- a so called Shared Memory, directly accessible by all the SPs of the MP

- a Constant Cache and a Texture Cache, managed by the framework

- a set of exclusive Registers for each SP

The Shared Memory can be considered as both a sort of "cache" of the Global Memory directly administrated by the cores in the MP and a mechanism for communicating among SPs belonging to the same MP. The Constant Cache and the Texture Cache are L1 caches used to speed-up the access time of the Constant Memory and the Texture Memory, respectively. Due to the fact that Constant Memory and Texture Memory are read-only (from a GPU point of view), no cache coherency protocol is required.

## 3.3   Programming Model

As stated at the beginning of this chapter, the CUDA framework enable the programmer to take advantage of the high number of parallel threads executable by the GPU to expoit TLP: ideally the program is organized into identical sub-problems - working on different data - that can be solved independently. Each of this sub-problems, called Kernels, is mapped onto a thread that will be executed on a SP.

All the threads executed by the SPs of a single MP are logically organized into a structure called Block. Virtually speaking, all the Threads belonging to a Block are executed in parallel. Usually, the number of SPs in a MP is much less than the number of Threads in a Block. For this reason, only a subpart of the Threads is in concurrent execution at a given time - the so called Warp. When a MP is loaded with a Block, it partitions it into warps that get sheduled by a warp sheduler for the execution on that MP. Due to the fact that all the Threads of a Block are executed on the same MP, it should be noticed that they share all the MP resources (i.e. Shared Memory and Caches) and that they can be synchronized using a specific API barriers. All the Blocks are grouped into another logical structure called Grid. Considering that the number of Blocks of the Grid is usually greater than the number of MPs, not all the Blocks can typically be sheduled at the same time. Since the blocks are unordered, they can execute equally well on a GPU that can handle one block at a time and on one that executes a dozen or a hundred at a time, as a demonstration of the scalability offered by the framework. In order to avoid complicating the Block scheduling process, no extra-block threads syncronization mechanism is provided.
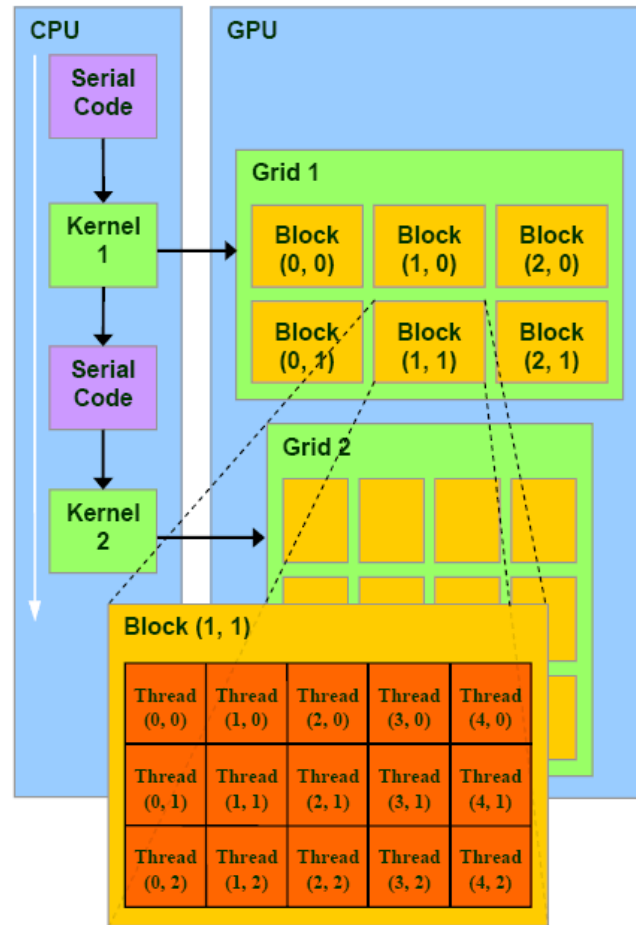
Figure 3.2: nVidia Execution Model, an example of heterogeneous programming.

## 3.4 Best Practices

In order to best exploit the resources of a CUDA capable GPU, there are several practices that must be adopted. Even if many of these are strictly dependent on the specific device used, there are some general rules that can be easily identified:

**Memory transfers** between Host and Device must be reduced to the minimum. Ideally there should be only one transfer Host-Device to store on the GPU the data for the computation and one transfer Device-Host to collect the results.

**GPU Occupancy** must be maximized. That is, the number of warps actually in execution must be closer to the maximum number of "in-fly" warps supported by the Device.

**Divergent Branches** should be avoided. Due to the fact that all the threads of the warp must execute the same instruction, in case of divergent branches the warp execution will be serialized, reducing the performance.

These three good pratices, ordered by decreasing importance, could be considered the most important rules that must be observed to effectivly developing in CUDA. It should be noticed that in the past this list was expanded by at least one element, coalescence. In the first CUDA Devices - identified by a so called "compute capability" berween 1.0 and 1.1 - all the threads in

the same warp had to access the memory words in sequence and, consequently, to avoid multiple reads/writes. Only if this pratice is enforced the MP can reduce to the least the number of memory accesses needed to provide the data to all the SPs. With modern devices - compute capability grater or equal to 1.2 - this constraint is much more relaxed: threads can access any word in any order, including the same words, and a single memory transaction for each segment addressed by the warp is issued. In this cases the problem of coalescence can be completely forgotten.

# Chapter 4

# Implementation

## 4.1 The original tool

The original tool is a Graphical Integrated Development Environment called QCADesigner, which is both meant for easing the process of layouting a QCA cells based circuit and for simplyfing the process of simulation and collection of the results. The layout view allows to place on the virtual PCB any number of QCA cell and defining if it is an input, output or fixed one. This discrimination makes it possible for the system to understand of which cells it is meaningful to record the polarization (the output cells) and to which one it is meaningful to assign a value (the input cells). Two different ways of simulating the behaviour of the circuit are implemented: the Coherence Vector and the Bistable engines.

Both realize the abstract notion of Cellular Automata: the status of every cell at a given instant $t$ strictly depends on its state and on that of its neighborhood at $t - 1$. The reason why two engines exist is quite simple: Bistable simulates the behaviour of the circuit from a "logical" point of view, meaning that it basically "switches" the polarization of every cell based on the polarization of the neighborhood and the current clock in *just one step*. Coherence Vector, instead, calculates at each instant $t$ the current polarization of the cell and that of its neighborhood and integrates over time the differential equations describing this system in order to make it evolve. Their authors compare and contrast these two engines this way: "...It is believed that although this approximation [that of Bistable engine] is sufficient to verify the logical functionality of a design it cannot be extended to include valid dynamic simulation; but, as a result of its simplicity this simulation engine is able to simulate a large number of cells very rapidly, and therefore provides a good in-process check of the design. For dynamical simulations refer to the coherence vector simulation. ". A few number of parameters allow for the customization of the behaviour of both engines in order to make the simulation as accurate and fast as possible. A few rules of thumb are available on the web site of the MiNa Group in order to effectively set some of them.

In the original implementation, moreover, it is possible to look at the evolution of the polarization of every cell during the simulation process.

## 4.2 The bottlenecks of the original implementation

First of all let's make a basic asymptotic analysis of the implementation of the Coherence Vector algorithm: we have $n$ cells that can have an average of $b$ neighbours. The engine basically runs trough every cell (this goes as $O(n)$) in the design and compute the current polarization based on the polarization of the previous step of all the neighbours (this goes as $O(b)$), and this is done for any possible combination of the inputs (this goes as $O(2^i)$). That is an asymptotic time complexity of $O(2^i * n * b)$ (worst case: $b=n-1$) and a space complexity of $O(n * b)$ (only the vector of the polarizations and the vector of neighbors of every cell must be saved, anything else is computed using these values).

Now, the main bottleneck of the original implementation derives from the *serialization* of the simulation of a system that could be actually evolved *in parallel*, both with respect to $i$ (eg: one thread per configuration of the inputs) and with respect to $n$ (eg: one thread per cell per iteration). The original simulation of the system in this serial fashion, moreover, introduces some additive error (even though distributable over all cells randomizing the order in which they are simulated at each iteration) due to an overlapping of the writes related to different time instants.

Our focus is set, in particular, on the Coherence Vector engine. The evidence of the limitations of the bottlenecks, here, are more evident than in Bistable because there is not not even the stabilization phase that would require additional costly synchronization mechanism/convergence criteria at each iteration of the simulation. In Coherence Vector mode the system is always free to evolve.

Before starting talking about the improved version it is good to remember that a number of runs of a set of test circuits has been made with QCADesigner in order to establish the baseline for the evaluation of the progresses (measured in seconds necessary to run the entire simulation).

## 4.3 Parallelization in CUDA

### 4.3.1 The approach

The guiding idea during the parallelization process is: any cell in the system evolves independently with respect to the evolution of the other cells at the same time, but only on the previous state of the machine. Moreover, any input configuration is totally independent on every other possible configuration. In line of principle, this could be approached in one (or both) of these two ways:

- parallelize with respect to $i$: spread the computation of all the possible $2^i$ configuration on that many threads.

- parallelize with respect to $n$: spread the computation of all the cells belonging to the same iteration on that many threads.

Our choice has been to follow the latter approach, mainly because

- major limiting factor of the former approach: it is feasible only for combinatoric circuits; sequential ones still need to consider the variation of the inputs in order to determine

their dynamic. Simulating a sequential circuit by means of a single input is practically useless.

- the "one cell-one thread" approach is easy both to figure out and to manage

- parallelizing with respect to the number of inputs would make the single kernel *very* complex, disallowing a number of possible optimizations (eg:occupancy improvement, branch divergence)

- not the very same actions should be taken for every cell at any moment of the simulation of different inputs; this introduces a hurdle in terms of branch divergence that doesn't allow the GPU to fully exploit its resources

- suppose to be simulating $k$ configurations of the inputs at a time. The spatial complexity of the algorithm increases by $k$ (which grows exponentially with the number of inputs!).

Moreover, the second approach allows us to better exploit the SIMD principle proper of all CUDA enabled devices because inside the same iteration of the simulation the actions to be carried over any cells are quite the same (and those cells who are not complying with this general scheme don't induce particularly critic divergences in the kernel). This approach allows for the same instructions to be concurrently executed on different data. In our particular case, the code basically does the same actions of the original implementation, apart from the details regarding the proper distribution of the computation among the available resources.

Last but not least, no *tile-problem* exists (the problem of overlapping adjacent concurrently simulated areas of circuit) because every single cell evolves independently of the effects of the neighbours at the *same* time $t$.

### 4.3.2 The implementation

Now, the datils of the implementation.

After an initial phase of parsing the command line commands, the flow of the program goes into Bistable or Coherence Vector engine.

In Coherence Vector mode, three different things happen.

First, all the useful data structures are initialized (for the output, for the parameters of the solver, for the iteration over all the cells into the design, notably). Among the others, some structures we had to implement in order to ease the process of converting the orignal data to CUDA compatible ones. In particular, we extract the following data:

- *polariztion vector* the vector of the initial polarization of every cell (size: $n$)

- *neighbour matrix* the unique index associated to every cell is found in this vector in position $i$ if and only if that cell is a neighbour of cell of uinique identifier $i$ (size: $n * b$)

- *clock value* a value between 0 to 3 representing the phase of the clock (size: $n$)

- *kink energy matrix* the element in position x,y contains the double representing the kink energy between element x and element y, and it is fixed troughout the whole simulation. (size: $n * b$)
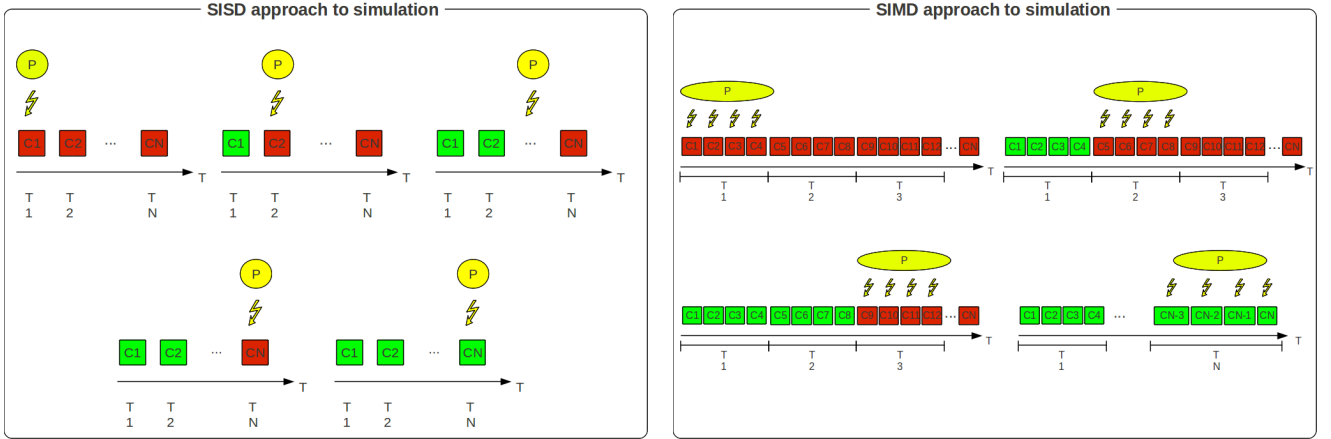
Figure 4.1: In the first approach (SISD) each cell is evolved into the new state *one at a time*. Observing that any cell inside the same iteration can be evolved *in parallel* allows us to exploit a SIMD architecture in order to replicate the same instructions over a large number of cells at a time.

- *next polarization vector* each element of this matrix represents the evolution of the state of the corresponding cell and is computed by an apposite kernel function (size: $n$)

These data are represented in Figure 4.2

Second, a number of calls to the framework APIs are made in order to load all the necessary data to the GPU. During this phase the geometry of the blocks of threads is also defined.

Third, the main iteration begins its execution. This loop is executed $f$ times, one for each instant required to completely simulate all the possible combinations of the inputs with the given time sampling (which is indeed a parameter settable via the command line). During each of the $f$ iterations all the $n$ cells in the design are evolved. This is basically the same happening in the serial code with the exception that we are capable of launching *one thread per cell*, each of which is able to calculate autonomously the evolution of the cell and storing this result in the corresponding cell of the **NP** vector (refer to Figure 4.2).

This part of the algorithm is the core of the speedup of our solution. What we do here, in contrast with the SISD approach of the original version of Coherence Vector, is exploiting a SIMD architecture in order to run $T$ (truly parallel) threads computing the evolution of $T$ different cells at once, without compromising the correctness of the simulation.

At the end of each iteration, the **NP** vector is transferred back from device to host memory and the polarization values written back into the original structures in order to be further treated, for example for saving to the output structures. These steps could have been (partly) done in GPU, but the cost/benefit ratio has been estimated too high to be convenient improving this part of the code. After this phase, the iteration goes to instant $t + 1$, the polarizations are loaded into device memory and the evolution kernel is launched once again.

Our approach, moreover, reduces the overall simulation error with respect to the CPU one: in fact, the evolution of the single cell in CPU mode is directly overwritten inside the *current* state, thus leading to the undesired effect of computing the state of the neighbour of $x$ with a polarization value that is already the future one. This is a limitation known to the MiNa Group who preferred to save execution time (otherwise to be dedicated to costly memory
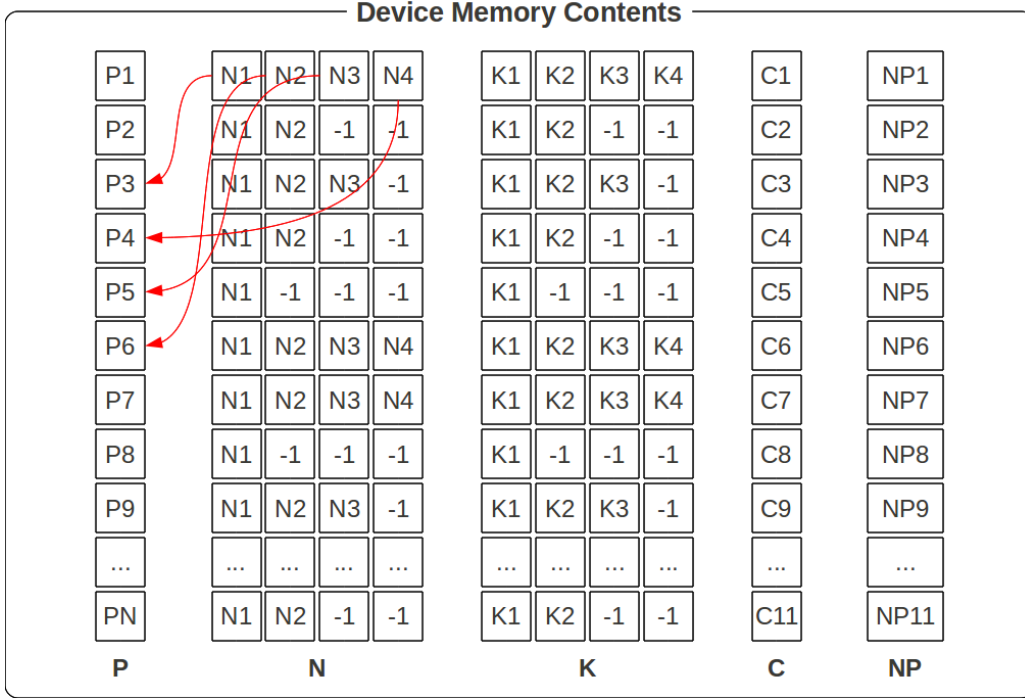
Figure 4.2: The contents of the memory. **P** and **NP** are, respectively, the vector of current polarization and the vector of next/future polarization. **N** is the matrix containing the "pointers" (in red) to the neighboring cells. A $-1$ in any position means that there are no more neighbours. This is because not all the cells must have the same number of neighbors and the matrix must have fixed dimensions while no value can be left unset. **K** matrix, instead, represent the kink energy that is instaurated between the corresponding neighbouring cells. **C** vector contains the clock value (the pahse of) for the corresponding cell.

allocations/deallocations) in favour of a slightly more imprecise simulation.

After these improvements, the complexity of the algorithm changes in the following way. From a time perspective we have a theorethical reduction by $T$ running threads; in reality it is slightly less due to the inveitable serializations of the threads concurrently accessing the polarizations of common neighbors in the **P** vector and only under the assumption that the (time spent loading data to/from device/host)/(time spent simulating the circuit inside the GPU) ratio is negligible. This is similar to what happens with loop unrolling + software pipelining. From a space perspective, instead, we obtain (refer to Figure 4.2) a complexity of $O(3*n + 2*n*b) = O(n(3 + 2*b))$ (worst case: $b = n - 1$).

### 4.3.3 Optimizations

A number of optimizations have been taken into account for improving the overall performance of the program.

- *memory hierarchy exploitation* memory in CUDA is higly hierarchical. Global memory is the main and largest memory pool of the device, at the cost of the slowest access speed. For this reason, whenever possible, it has been tried to move data to constant memory (e.g. all the optimization dependent constant values, pre calculated ones etc).

No shared memory can be effectively shared between threads because the neighbours of a cell are not necessarily the same neighbours of another thread in the same warp. Other examined solutions comprised more or less huge (read: low speed) transfers of data from host to device memory, impacting at once both on the size of the largest possible simulated circuit and a huge amount of overhead not justified by a proportional in-kernel time. For example, instead of passing the uinique identifiers to the neighboring cells (the **N** matrix) one could directly pass the *value* of their polarization, allowing, for example, easier coalescent accesses. Even though, at first, this approach could be positively judged, at a deeper inspection it results in a predictable failure. In fact, every time the iteration ends we are required to exit from the kernel and overwrite the old polarization values with the new ones, thus *invalidating* the matrix passed just the step before. Our estimate is that a transfer of $n * b$ more doubles (64bit each in CUDA) at each iteration would impose a strong penalty to those circuits that are particulary well suited for simulation in GPU, *id* est the big ones.

- *register usage* registers in CUDA capable devices are a scarce resource that directly and strongly impact the overall performance of a kernel. In fact, the number of threads *truly* concurrently running depends on how many registers are available inside the streaming multiprocessor to support their execution. Even though it is possible to define what is the optimal geometry of the running blocks by hand, nVidia made available a simple spreadsheet that allows for a fast computation of the optimal parameters for the particular kernel running on the particular system. The term "occupancy" in this context refers to the efficiency at exploiting the resources and is defined, more precisely, as the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Obviously, the higher the occupancy, the higher the active warps, the faster the computation. In order to improve the occupancy factor we developed two different versions of the main kernel, one for each type of integration method employed fo the evolution of the polarization of the cells. This allowed us to reduce at once both the complexity of the single kernel (reducing the number of branches and the total number of instructions) as well as the number of registers required to run them (from 20 to 16, each 64 bits wide). This is actually the best ppossible solution under a Tesla C1060 (the actual card used for testing purposes), yelding, in fact, a 100% occupancy rate.

- *exploitation of fast_math* CUDA capable devices do not fully comply to the IEEE754 standard for floating point, double precision operations in that they allow a bigger ulp error for certain operations, in particular for trigonometric ones. In particular, the absolute value of the difference in ulps between a correctly rounded double-precision result and the result returned by the CUDA library function is 2 for cosine and 1 for the hyperbolic tangent. In our code we largely make use of floating point operations, so, by Amdhal's law, it is a good place where to look for improvements. One of those could be the demoting of the double arguments to float ones, in order to exploit the fast_math mode of the CUDA C math library. In this mode, cosine function and division operation take only one clock cycle to execute, allowing for the highest possible troughput under this architecture. [noi non l'abbiamo ancora provata, potrebbe essere interessante ma nn abbiamo tempo per testarla - la mettiamo? la mettiamo come possibile ma non testata per motivi di tempo?]

- *small transfers of data to/from host/device memory* these are by far the slowest possible operations that must be limited as much as possible. Our solution requires the lowest number of transfers among the proposed ones. Profiling of our code yield only a 9% of the execution time spent at this task.

# Chapter 5

# Results

In order to evaluate the performance of CudaQCAD, several tests were executed. These tests aimed to show strengths and fallacies of the simulator both in comparison with the original QCAD and in the usage of the resources offered by the architecture. In the next sections will be described the Hardware used for the tests and the organization of the tests themselves.

## 5.1   Hardware Description

All the tests were performend using the "Lucifero" Workstation, located in the "MicroLab" of Politecnico di Milano.

**CPU**  Intel Xeon E5345

**GPU**  Nvidia Testa C1060

## 5.2   Test 1: CPU vs GPU

The most meaningful test executed is certainly the comparison between the original QCADesigner and CudaQCADesigner. Both the versions of the simulator were tested on a pool of QCAs composed of several real circuits of different complexity. This test was designed in order to evaluate:

**Correctness** of the two simulators.

**Coherency** of the results.

**Performance** in terms of execution time.

Considering the first two points, the test has shown that both the simulators obtain the same results for all the circuits of the pool and that these results were aligned with the well known behaviour of the circuits.
With respect to the performance, it was noticed a considerable speed-up - 30x peak - of the GPU version over CPU version only when the number of cells composing the simulated circuits
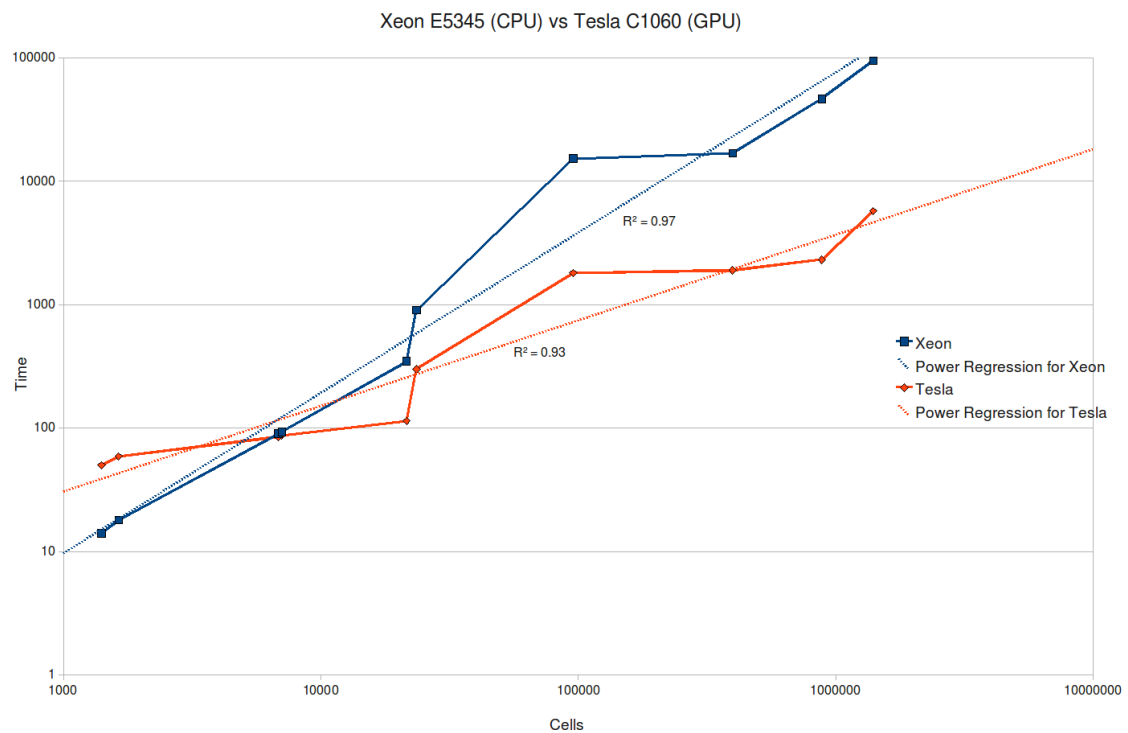
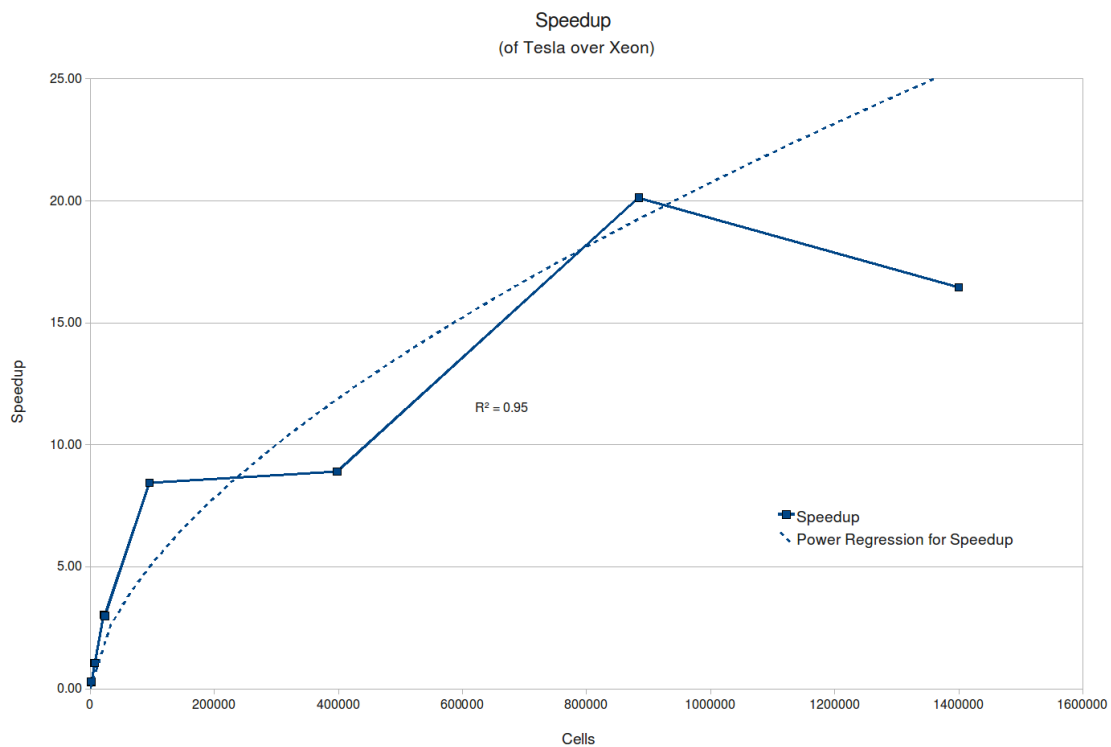Figure 5.1: Performance comparison between Intel Xeon and Nvidia Tesla.



Figure 5.2: Overall SpeedUp.

was greater than 7000. For circuits with a lower number of cells, the original QCADesigner outperform CudaQCADesigner, Fig 5.1.

This result was not surprising. Until the number of cells in the circuits is lower than a given threshold - 7000 with the used architecture - the overhead due to the device setup time is bigger than the gain due to the parallelism. However, considering that the original goal was to speed-up the simulation of those circuits that requires an amount of time too high to be considered acceptable for pratical purpouses on a SISD architecture, this test can be considered passed.

## 5.3 Test 2: Memory Transfers

As stated in chapter 4, memory transfers between host and device are one of the most important factors, and often the most limiting, in order to achieve good performance in CUDA.
A well built GPU algorithm should not waste more than 20% of the whole execution time for Host-Device I/O. Considering this assumption, the results achieved by CudaQCADesigner are actually good: all the tested circuits showed a memory transfer rate lower than 10%, even those under the threshold of 7000 cells (Fig 5.3, 5.4).
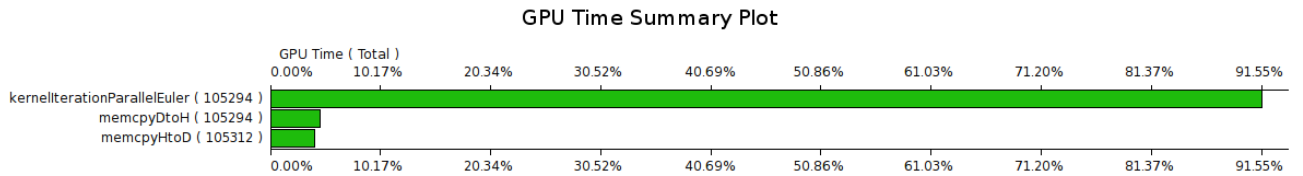


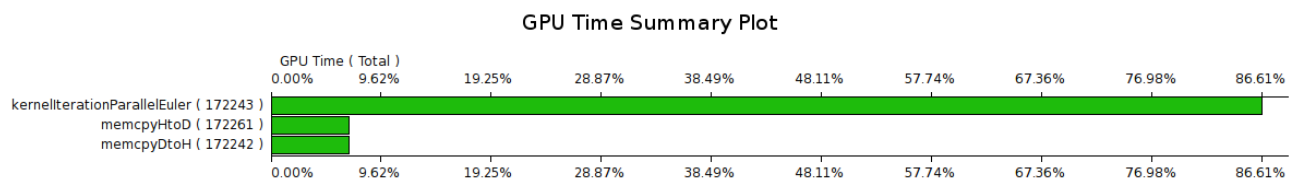Figure 5.3: Memory transfers for MUX42 circuit (21551 cells).



Figure 5.4: Memory Tranfer for NAND circuit (1642 cells).

## 5.4 Test 3: GPU Occupancy

This test aims to prove the effective usage of the GPU resources. The occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. As stated in chapter 3, each multiprocessor on the device has a set of N registers available for use by CUDA thread programs. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor. The Cuda compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine

simultaneously. If a program tries to launch a kernel for which the registers used per thread are too many, the number of threads in concurrent execution will decrease, and as a consequence, also performance will do.

Unfortunalely there is no way to automatic tune the execution parameters in order to achieve the full occupancy. For this reason, several attempts were performed and profiled, Fig 5.5. Considering that the occupancy can be fairy evaluated only when the amount of computations is big enough to require the usage of all the available resources, for this test a circuit with a number of cells greater than the threshold was choosen.

The first attempt was unsatifactory, due to an occupancy of only 50%. For this reason the block size was progressively enlarged till achieving the maximum occupancy supported. Unfortunately, even with a larger block size, the occupancy could not be moved to more than 75%, due to the register pressure.

In order to further improve the occupancy, the kernel function was re-designed with special attention to the number of registers per threads, as described in chapter 4. The new version of the kernel leads to much better results and the occupation was barought up to 100%.
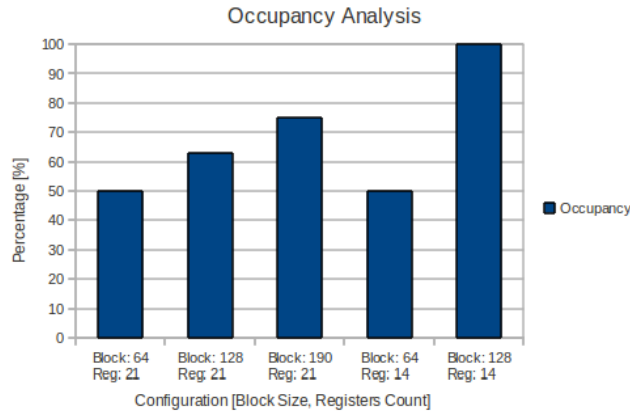


Figure 5.5: Occupancy Analysis. Data extracted from Nvidia Occupancy Calculator.

## 5.5 Test 4: Divergent Branches

The number of divergent branches can seriously impact on the performance of the simulator. As a rule of thumb the percentage of divergent branches must be lower than 15% to be considered acceptable. In all the performed test, CudaQCADesigner was actually under this threshold, with an average divergent branche rate of 9%.

# Chapter 6

# Conclusions

Our CUDA implementation allows for an efficient exploitation a nVidia's C1060 Tesla card for exhaustively simulating large and complex QCA cells based circuits. Our implementation has been tested against a number of circuits with the original QCADesigner algorithms and we have verified that the property of correctness holds.

Two main improvements are possible for this application

- the simulation mode called "vector table", that allows for a reduction of the $2^i$ term of the complexity formulae by manually disabling some of the inputs cells of the design

- the implementation of a (simple) mechanism to spread the load of the computation onto a *cluster* of Tesla cards. As in the single card case, the time complexity of the algorithm is reduced by the number of running threads; the speedup is then limited only by the number of nodes present in the cluster (which directly maps on the number of concurrently runnable threads, ie the number of concurrently simulable QCA cells).