Politecnico di Milano - University of Illinois Chicago
HPPS Project

QCADESIGNER POWERED BY CUDA

Instructor: Prof.ssa Sciuto Donatella
Tutor: Santambrogio Marco Domenico

Authors:
Gibilisco Giovanni Paolo, Matr. 755066
Marconi Francesco, Matr. 755439
Miglierina Marco, Matr. 754848

2009-2010

Contents

List of Figures

Abstract

2 righe sul lavoro, -¿VENDITI BENE

Chapter 1

State of the Art

1.1 Qunatum Dot Cellular Automata

During the last years the main effort of companies interested in developing information technology has been in the direction of shrinking size of transistors, the main component of any modern computer. This effort allowed to build machines able to run at very high frequency with a huge number of elements (memories, cores..) on a single chip. This trend can't go on forever. In the search for a new computational model we focused on Quantum Dot Cellular Automata because it's a very promising technology. It's base component is the Quantum Dot, a nanostructure that can be built with different materials and different technologies which is able to store a well defined number of electrons. Using 4 quantum dots it's possible to create a cell like the one in figure.

If we inject 2 electrons in this structure their reciprocal repulsion allow them to be only in two configurations, with the electrons disposed on one of the two diagonals. Electrons are free to move according to a clock signal but they can never leave the cell. The configuration in the picture above represents the logical bit 0 the opposite configuration is the bit 1. If we put

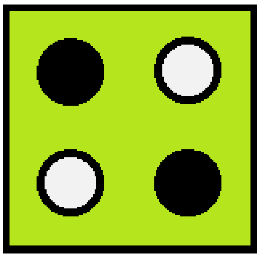
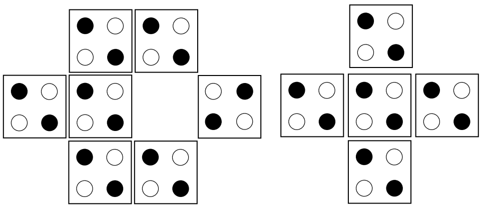


Figure 1.1: A Cell made of 4 Quantum Dots

two cells one beside the other we can see that the polarization of the first cells has an impact on the polarization of the other. This effect can be used to build complex logic circuits. Main blocks of QCA circuits are NOT and Majority Gate both shown in the figure below.

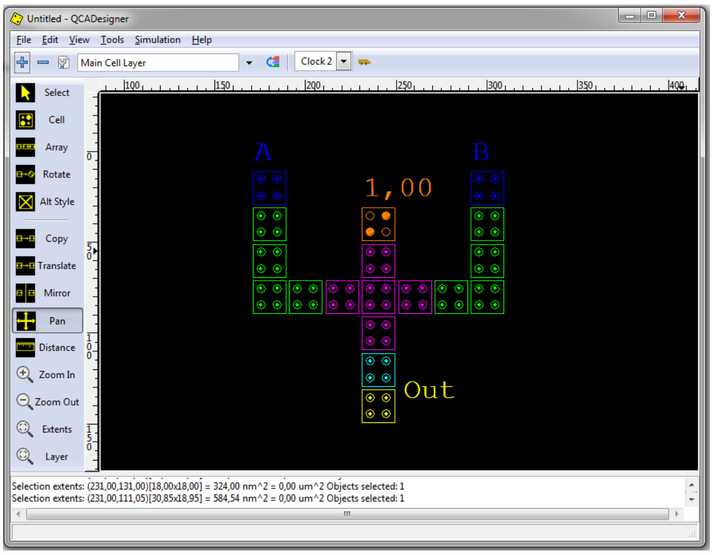


1.1.1 Clock

A key issue in the design of circuits is the definition of clock areas. This technology uses 4 identical clock signals which are shifted of $frac\pi 2$. Each clock signal has 4 clock phases: switch, hold, release and relax. During this fases potential barriers are raised or lowered inside the cell in order to contorl the movement of electrons. In the relax phase barrers are low and electrons can freele move inside the cell, during the switch phase barriers are rised and the movement of electrons are limiteted. during the hold phase eletrons aren't allowed to move within dots. In the relax phase the barriers are lowere. The alternation of clock phases identify the flow of information into the circuit.

1.2 QCADesigner

QCADesigner is a tool made by the University of British Columbia to design and verify circuits in QCA technology. This tool offers two main feature. The first is to design with an intuitive graphical user interface a circuit, in the image below is showed a simple circuit, the second is to simulate the behaviour of the circuit.



```
Selection extents: (231,00,131,00)[18,00x18,00] = 324,00 nm^2 = 0,00 um^2 Objects selected: 1
Selection extents: (231,00,111,05)[30,85x18,95] = 584,54 nm^2 = 0,00 um^2 Objects selected: 1
```

To verify the behaviour of the system two simulation engines are offered. The bistable simualtion and the coherence vector simulation. the aim of this report is to speedup the execution of the bistable simulation engine so we are going to brefly describe only this type of simulation. The coherence vector differs in the way the polarization of cells is calculated, it's much slower but gives more accurate result.

1.2.1 Bistable Simulation

Bistable simulation engine has been created in order to give designers a way to test their circuit very rapidly. For the simulation the cell is modeled as a two state system, for each cell the kink energy with repsect to neighbour cells and the polarization is stored. the kink energy takes into account the cost of two neighbouring cells with different polarizaions. At each step of the execution the new polarization of each cell is calculated on the basis of the old polarizations and the kink energy of all neighbouring cells, within a predefined radius. The approximations made by ths simulation engine is sufficient to verify the logical behaviour of the system but doesn't descirbe well it's dinamics. For the inspection of dynamic situations coherence vector engine should be used. The main advantage of this kind of simulation is that it's very quick with respect to the cohrence vector engine.

1.3 CUDA

In recent times, Graphics Processing Units (GPUs) have been considered a potential source of computational power for non-graphical applications, due to the ongoing evolution of their programming interfaces and their appealing cost-performance figures of merit. Recent works had first attempted to adapt general purpose applications to the graphic rendering APIs (OpenGL and DirectX), which up to two years ago represented the only interface to tap into the GPU computational resources. Tesla is NVIDIA's first dedicated General Purpose GPU.

1.3.1 NVIDIA Tesla Architecture

Modern GPUs include hundreds of processing elements. The NVIDIA Tesla GPU series provide a set of independent multithreaded streaming multiprocessors. Figure **??** shows an overview of the NVIDIA Tesla streaming processors array which is the part of the GPU architecture responsible for the general purpose computation. Each streaming multiprocessor is composed by a set of eight streaming processors,two special functional units and a multithreaded instruction issue unit (respectively indicated as SP, SFU and MT-Issue in Figure **??**).
A SP is a fully pipelined single-issue processing core with two ALUs and a single floating point unit (FPU). SFUs are dedicated to the computation of transcendental functions and pixel/vertex manipulations. The MT-Issue unit is in charge of mapping active threads on the available SPs.
A multiprocessor is able to concurrently execute groups of 32 threads called warps. Since each thread in a warp has its own control flow, their execution paths may diverge due to the independent evaluation of conditional statements. In this case, the warp serially executes each path. When the warp is executing a given path, all threads that have not taken that path are disabled. On the other hand, in case the control flows converge again, the warp may return to a single, parallel execution of all threads.
Each multiprocessor executes warps in a fashion much like the Single Instruction Multiple Data (SIMD) paradigm, since every thread will be assigned to a different SP and every active thread will execute the same instruction on different data.
The MT-Issue unit weaves threads into a number of warps and schedules an active warp for execution, using a round-robin scheduling policy with aging for this purpose.
Streaming multiprocessors are in turn grouped in Texture Processor Clusters (TPC). Each TPC includes three streaming multiprocessors in the Tesla architecture. The TPC also includes support for Texture processing, though these features are seldom used for general purpose computing and will not be investigated in this description.

Finally, the NVIDIA GPU on-board memory hierarchy includes registers (private to each SP), on-chip memory and off-chip memory. The on-chip memory is private to each multiprocessor, and is split into a very small instruction cache, a read-only data cache, and 16 KB of addressable shared data, respectively indicated as I-cache, C-cache and Shared Memory in Figure **??**. This shared memory is organized in 16 banks that can be concurrently accessed, each bank having a single read/write port.
The GPU we used for our project is the NVIDIA Tesla c1060. With the computational power of its 240 Streaming Processors (grouped into 30 TPCs) and the 102,4 GB/s max bandwidth of its 4096 MB GDDR3 memory, it represents one of the most performing GPUs on the market.

1.3.2 CUDA programming model

The Compute Unified Device Architecture (CUDA), proposed by NVIDIA for its GeForce (8 series and above), Quadro and Tesla graphics processors, exposes a programming model that integrates host and GPU code in the same C/C++ source files.
The main programming structure supporting parallelism is an explicitly parallel function invocation (kernel) which is supposed to be executed by a user-specified number of threads. Every kernel is explicitly invoked by host code and executed by the device, while the host-side code continues execution asynchronously after instantiating the kernel. The programmer is provided with a specific synchronizing function call to wait for the completion of the active asynchronous kernel computation.
The CUDA programming model abstracts the actual parallelism implemented by the hardware architecture, providing the concepts of block and thread to express concurrency in algorithms. A block captures the notion of a group of concurrent threads. Blocks are required to execute independently, so that it has to be possible to execute them in any order (in parallel or in sequence). Therefore, the synchronization primitives semantically act only among threads belonging to the same block. Intra-block communications among threads use the logical shared memory associated with that block.
Since the architecture does not provide support for the message-passing techniques, threads belonging to different blocks must communicate through global memory. The global memory is entirely mapped to the off-chip memory. The concurrent accesses to logical shared memory by threads executing within the same block are supported through an explicit barrier synchronization primitive.
A kernel call-site must specify the number of blocks as well as the number

of threads within each block when executing the kernel code. The current CUDA programming model imposes a capping of 512 threads per block.

The mapping of threads to processors and of blocks to multiprocessors is mainly handled by hardware controller components. Two or more blocks may share the same multiprocessor through mechanisms that allow fast context switching depending on the computational resources used by threads and on the constraints of the hardware architecture. The number of concurrent blocks managed by a single multiprocessor is currently limited to 8.

In addition to the logical shared memory and the global memory, in the CUDA programming model each thread may access a constant memory. An access to this read-only memory space is faster than one to global memory, provided that there is sufficient access locality since constant memory is implemented as a region of global memory fit with an on-chip cache. Finally, another portion of the off-chip memory may be allocated as a local memory that is used as thread private resource. Since the local memory access is slow, the shared memory also serves as an explicitly managed cache though it is up to the programmer to warrant that the local data being saved in shared memory are not accessed by other threads. Shared memory comes in limited amounts (threads within each block typically share 16 KB of memory) hence, it is crucial for performance that each thread handle only small chunks of data.

1.3.3 SIMT and SIMD

SIMT architecture is similar to single instruction, multiple-data (SIMD) design, which applies one instruction to multiple data lanes.

The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMD instruction controls a vector of multiple data lanes together and exposes the vector width to the software, whereas a SIMT instruction controls the execution and branching behavior of one thread.

In contrast to SIMD vector architectures, SIMT enables programmers to write thread level parallel code for independent threads as well as data-parallel code for coordinated threads.

For program correctness, programmers can essentially ignore SIMT execution attributes such as warps; however, they can achieve substantial performance improvements by writing code that seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional codes: programmers can safely ignore cache line size when designing for correctness but must consider it in the code structure when designing for peak performance.

SIMD vector architectures, on the other hand, require the software to manually coalesce loads into vectors and to manually manage divergence.

Chapter 2

Rationale

- Why QCA? * novel emerging paradigm * 2 Thz, low energy consumption and miniaturization * quantum computing - QCADesigner simulator slow on big circuits: * Every sample, each cell's polarization is computed based on the values of his neighbors, sequentially. * Bottleneck from profiling (table with times) * Simulation core: pseudo code * Identical operations repeated for each cell, big circuits -¿ thousands of cells * We chose to speedup this part of the code with CUDA because: SIMD architecture (SIMT): single instruction repeated on different data hundreds of core -¿ many threads running simultaneously, each thread responsible of computing a cell's polarization scalable, adding new cores implies a greater number of cells computed simultaenously, higher speedup - Objective * Speed up simulation for big circuits * batch simulator * Given a file .qca -¿ produce output: binary, continous values, plot on png, log with info of simulation * if same .qca -¿ same results CPU and CUDA

Chapter 3

implementation

3.1 First approach

The original source code of QCADesigner was downloaded from Mina website (ref). We attempted to make it compile as it was but we did not manage to solve several compilation errors. So we started to focus on the identification of the core algorithm supporting the tool in order to obtain a working batch simulator executable on CPU. This operation took us some weeks of work. Meanwhile we were able to deeply analyze the code. We made some hypothesis on the location of possible bottlenecks, we identified the data structures used to represent circuits and started to consider possible transformations that had to be done in order to obtain fast accessible and light weight data structures allocable on the GPU global memory.

3.2 CPU algorithm and profiling

Bistable engine is thought as a fast and approximated simulation, sufficient to verify the logic functionality of a design. Every cell is represented as a simple two-state system. The entire simulation is divided into samples, that are units of time (not yet experimentally defined), and for each sample the state of each cell is calculated with respect to the other cells within a preset effective radius. This operation is iterated until all cells have converged within a predetermined tolerance. Once the entire system converges the output is recorded and the computation goes on with the next sample, after having updated the input cells with new input values. The number of samples required to have a good approximation is known (ref) to be $2000 * 2^N$, where N is the number of inputs. The maximum number of iterations allowed for the convergence within a sample is 100. Thus, during a simulation each cell's value is computed sequentially $It * 2000 * 2^N$ times, where It is the mean number of iterations needed to reach convergence. The more are the cells, the longer will take each sample to reach convergence.

Once we finished the batch application we started to profile the execution times simulating some circuits. The table (ref) shows

3.3 Parallelization Strategy

3.3.1 Data Structures

In the CPU implementation all the informations about circuit structure and cells' details are stored in complex nested structures: considering CUDA SIMT parallelism, well suited to operate over structures like matrices anda arrays, we first detected the useful data for simulation on GPU and then decided how to organize them for a fast CUDA implementation.
The essential data we identified were: cells' polarizations, neighbours'lists, intra-cells kinetic energy (eK) values, clock values, stability status and other predefined constants.
We decided to store most of these values (the ones that) in simple array structures, in order to obtain a clearer thread mapping and a faster thread access.
Cells' polarizations array is in charge to contain the polarization values for all the cells through all the iterations, providing the results for simulation output. At each iteration, polarizations are updated by threads.
Neighbours' and eK arrays contain all the informations we need about circuit structure and energy interactions among the cells. They fully depend on circuit's geometry, and don't change during the simulation. However, they are essential to calculate cells polarizations.
Stability array contains simple boolean values wich provide the stability status of every cell and permits the evaluation of array global stability after every iteration.
Other significant structures we designed in our implementations are the input and output cells' indexes arrays, two auxiliary structures which help us respectively to update input cells at the beginning of every sample and to store output cells' polarization at the end of every iteration.

3.3.2 Parallel Algorithm

As reported previously, in our parallal implementation we focused our attention on the simulation core of the algorithm.

Our goal was to exploit the parallel thread execution to compute simultaneously all the circuit's cells.

BISOGNA SPIEGARE LA ROBA DI KONRAD CHE CI HA FATTO PENARE, MA POI NE SIAMO USCITI ABILMENTE CON IL MIGLIE'S COLORING

After the initial circuit file reading and structures filling phases, we introduced a conversion function to have data in our desired format. After conversion the coloring algorithm is applied and all the needed data are ready to be allocated on the device.
Parallel bistable simulation, as showed in (ref algoritmo) could be divided in stages below:

- allocation and copy of the variables on the device, which is performed only one time during the execution.
- update inputs kernel, executed at the beginning of each sample to update inputs.
- bistable kernel, invoked in every iteration as many times as the number of colors in the circuit
- stability checking, performed after each iteration
- copy-back of output cells polarizations from device to host, necessary to save circuit output at the end of each sample
- cleaning device memory after the last sample.

SI POTREBBE FARE UN BELL'ALGORITMO SCRITTO BENE SULLO PSEUDOCOICE DELL'IMPLEMENTAZIONE.. NEL FILE CHE HO INCLUSO CI SONO DUE ESEMPI TRATTI DALLA MIA TESI

1em

3.3.3 Memory allocation

As reported in the section References, when engeneering an algorithm for the GPUs, a critical design decision is the allocation of data onto the different memories provided by the architecture. Almost all the structures above have changeable sizes, depending on the circuit structure (number of cells and neighbourhood among them). For that reason array dimension may ramp up to many thousands of elements, dramatically increasing the space needed on the device. While Tesla c1060's 4GB global memory is safely enough to store these structures, the same cannot be stated for the shared and constant

Algorithm 3.3.1: XTS-Serpent per cifrare una *Data Unit*

```
for $q \leftarrow 0$ to $m - 2$ do
    $C_q \leftarrow$ XTS-Serpent-blockEnc $(K, P_q, i, q)$;
endfor
$b \leftarrow$ dimensione in bit di $P_m$;
if $b = 0$ then
    $C_{m-1} \leftarrow$ XTS-Serpent-blockEnc $(K, P_{m-1}, i, m - 1)$;
    $C_m \leftarrow$ vuoto;
else
    $CC \leftarrow$ XTS-Serpent-blockEnc $(K, P_{m-1}, i, m - 1)$;
    $C_m \leftarrow$ primi $b$ bits di $CC$;
    $CP \leftarrow$ ultimi $(128 - b)$ bits di $CC$;
    $PP \leftarrow P_m|CP$;
    $C_{m-1} \leftarrow$ XTS-Serpent-blockEnc $(K, PP, i, m)$;
endif
$C \leftarrow C_0| \ldots |C_{m-1}|C_m$;
```

memories: these have very limited storage space and need to be exploited very carefully. We decided to use shared memory to store relatively small structures such as the input and output indexes arrays. The input array is used in the *update input kernel* to fastly find the input cells to be updated. In the same way, output indexes array is exploited in the *bistable kernel* to store output cells' polarizations.
The reasons of such a choice consist both in the limited sizes of these structures and in the high frequency of accesses to them during execution.
In the *constant memory* cache we allocated all the necessary variables which not vary during the execution and need to be accessed by all the threads, such as clock constants, number of cells (input, output and total amount), stability tolerance, number of samples maximum number of neighbours. All the remaining variables are stored in thread's local registers.

3.3.4 Optimizations

*shared memory (GIÃ CITATA) *coalescence (HO FATTO UN PROFILING, SUL CIRCUIT$_{20}4ESIVEDECHECISONOUNBELPO'DIACCESSINONCOALESC$

Algorithm 3.3.2: Algoritmo XTS-Serpent per Truecrypt

```
Input: $P = (P_d, P_{d+1}, \dots, P_{d+t})$: il plaintext chunk di ingresso diviso
       in una sequenza di $t$ data units.
       $[d, d+t]$: range degli indici di data units all'interno del
       plaintext che devono essere cifrate attraverso molteplici
       chiamate dell'algoritmo.
       $k_1$: chiave di cifratura primaria. $k_2$: chiave di cifratura
       secondaria.
Output: $C = (C_d, C_{d+1}, \dots, C_{d+t})$: il blocco di ciphertext, risultato
        della criptazione.
$E(p, k)$: cifratura simmetrica del plaintext $p$ tramite la chiave $k$.
$t = 512$: numero di data units presenti in un blocco di plaintext.
$m = 32$:numero di cipher blocks per ogni data unit.
$P_n = (P_{n_0}), \dots, P_{n,m-1}$,
$C_n = (C_{n_0}), \dots, C_{n,m-1}$ $\forall n \in [d, d+t]$.
$\bar{n}$ indice cifrato della data unit $n$
begin
    for $n \leftarrow d$ to $d + t$ do
        $\bar{n} \leftarrow E(n, k_2)$;
        for $i \leftarrow 0$ to $m - 1$ do
            $w_{n,i} \leftarrow \bar{n} \otimes \alpha^i$;
            $C_{n,i} \leftarrow E(w_{n,i} \oplus P_{n,i}, k_1) \oplus w_{n,i}$;
        endfor
    endfor
    return $C$;
end
```

Chapter 4

Results

Chapter 5

Conclusions

Bibliography