Politecnico di Milano - University of Illinois Chicago
HPPS Project

UIC

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

- QCA * QCADesigner * Two Engines: BISTABLE and COHERENCE, describe them shortly (see MINA site) -¿ BISTABLE IS JUST A FAST APPROXIMATION to test circuits

1.1 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 âgeneral purposeâ 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.

1.1.1 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.

1.1.2 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++ 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 â 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.

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

Chapter 4

Results

Chapter 5

Conclusions