Chimeh, M., Hall, C., and O'Donnell, J. (2012) *Optimisation and parallelism in synchronous digital circuit simulators*. In: IEEE International Conference on Computational Science and Engineering, 5-7 Dec. 2012, Nicosia.

Copyright © 2012 The Authors.

http://eprints.gla.ac.uk/70983/

Deposited on: 15 November 2013

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

Optimisation and Parallelism in Synchronous Digital Circuit Simulators

Mozhgan Chimeh
School of Computing Science
University of Glasgow
Glasgow, UK, G12 8QQ
Email: mozhgan@dcs.gla.ac.uk

Cordelia V. Hall
School of Computing Science
University of Glasgow
Glasgow, UK, G12 8QQ
Email: cordelia.hall@glasgow.ac.uk

John T. O'Donnell
School of Computing Science
University of Glasgow
Glasgow, UK, G12 8QQ
Email: john.odonnell@glasgow.ac.uk

***Abstract*—Digital circuit simulation often requires a large amount of computation, resulting in long run times. We consider several techniques for optimising a brute force synchronous circuit simulator: an algorithm using an event queue that avoids recalculating quiescent parts of the circuit, a marking algorithm that is similar to the event queue but that avoids a central data structure, and a lazy algorithm that avoids calculating signals whose values are not needed. Two target architectures for the simulator are used: a sequential CPU, and a parallel GPGPU. The interactions between the different optimisations are discussed, and the performance is measured while the algorithms are simulating a simple but realistic scalable circuit.**

***Keywords*-synchronous digital circuit; circuit simulation; optimisation; parallelism; GPU**

I. Introduction

Digital circuit simulation is essential to the successful design of integrated circuits, including the central components in computer and mobile devices. However, modern circuits can be large, containing thousands to millions of components. Furthermore, every component may be active on every clock cycle, and it may be necessary to simulate the circuit for many clock cycles. Therefore a full circuit simulation may require a long execution time.

This paper describes several techniques for optimising circuit simulators, discusses the way these optimisations interact with each other, and presents experimental work that assesses the techniques. There is no one algorithm that is best for all situations; the performance of a simulation algorithm depends on the architecture of the circuit being simulated and the characteristics of the input data given to that circuit. These results presented here are helpful in selecting a simulation algorithm that is appropriate for a particular problem. The results will also help to guide future research in high performance simulation algorithms.

Circuit simulation is a family of related problems, because there are many different kinds of information that may be required from simulating a particular circuit. In this paper, we consider only synchronous circuits with a single clock. Asynchronous circuits are also important, especially for very large chips, but these are often organised as networks of synchronous circuits.

Two target architectures are used for running the simulations: a sequential CPU programmed in C, and a parallel GPGPU programmed in C+CUDA [1]. Another useful target, a multicore processor programmed in OpenMP, or a similar system, is not considered in this paper. A GPGPU is a general purpose Graphics Processing Unit; this is a specialised multicore processor that can execute a large number of threads in parallel in a data parallel style. CUDA is an extension to C for programming GPUs manufactured by NVidia.

We now introduce some terminology about the application. A circuit has a number of inputs and outputs. These values, and generally all values on wires, are called signals. A circuit operates through a sequence of clock cycles, and on every clock cycle a value is read from each input signal and is written to each output signal.

A circuit simulator is software that predicts the behaviour of a circuit. It is much faster and less expensive to simulate a design than to fabricate it in real hardware and test that. A simulator takes two major inputs: (1) a netlist that describes precisely the structure of the circuit, and (2) the values of the input signals for every clock cycle. A netlist is a representation of the circuit graph, defining all the components and their interconnections via signals (i.e. wires). Normally, the netlist is produced by a hardware description language that allows the designer to specify the circuit in a readable notation. In this work we use circuits specified by Hydra, a functional hardware description language [2] [3] [4].

Section II discusses related work. Section III describes the test circuit used to evaluate the simulation algorithms, which are explained in Section IV. Section V discusses the performance of the algorithms, and Section VI concludes.

II. Related work

The structure and behaviour of digital circuits are explained in [5]. Simulation algorithms for digital circuits must obey the relevant aspects of the circuit model in use (for example, in this paper we consider only the synchronous model) [6] [7]. One approach is to write a general simulator that reads in a netlist for an arbitrary circuit, then reads the input signal values, and calculates the output signals [8]. An alternative approach is to read in a circuit specification and translate it into a simulator which is specialised for that circuit [9] [10].

There is increasing interest recently in using GPUs to speed up circuit simulation [11] [12] [13] [14] [15] [16]. Our work incorporates many of the techniques presented in these papers, but our main focus is on assessing the way that different optimisations interact in a family of closely related simulators.

III. A target circuit: Register transfer machine

The performance of a simulation algorithm depends on the nature of the target circuit that is being simulated. The most important circuit properties include the critical path depth, the number of logic gates at each path depth, the number and complexity of different component types that occur in the circuit, and the presence of special circuit features that allow specific optimisations. Real circuits vary widely in all these characteristics.

To assess the optimisations discussed in the following section, we want (1) a circuit that is simple yet exhibits typical characteristics, and (2) a way to generate a family of similar circuits with increasing numbers of components. Our approach is to start with a realistic circuit called the RTM, which is described below, and to generate larger circuits by replication.

The basic RTM circuit contains 22 logic boxes (at a higher level than logic gates), and 126 wires. It has a critical path depth of 11, excluding registers and input/output components. An input component contains the group of input signals required by the circuit and the output component takes the group of signals being displayed or probed.

To study simulator performance as circuit size grows, we generate larger circuits that comprise N copies of the RTM; thus the sizes of the input vectors, numbers of components at each path depth, and size of output vectors all grow linearly with N, but the critical path depth remains constant. The initialisation and input/output portions of the algorithms take account of the N copies, but the simulation algorithms are all written in a general form and do not exploit any knowledge that the larger circuits are actually replications of a smaller one.

An important point is that in real circuits, the critical path depth does *not* grow linearly (or in any other predictable way) with the number of components; indeed, many realistic techniques for high performance processors use many extra logic gates in order to *reduce* the critical path depth at the expense of more logic gates at each path depth.

The RTM circuit is a register transfer machine. There is an internal register file with two output ports that are connected to a ripple carry adder. The data input to the register file is determined by a multiplexer that can select either the output of the adder or an external input.

Although it is a small circuit, the RTM is the kernel of a RISC processor architecture, and it illustrates several characteristics of real circuits. Furthermore, the RTM is programmable, as its behaviour depends on the settings of its control inputs. Thus the behaviour of the circuit depends strongly on the input values—not just on what numbers are calculated, but on what operations are performed. Some of the optimisations described in the next section exploit these characteristics. Therefore we use three distinct programs (expressed as control input settings) that cause the RTM to exhibit different types of behaviour. These programs have varying proportions of load and add operations.

TABLE I
Program A simulated by the RTM

data	add	load	dreg	clr	sreg1	sreg0	ci
0,0,0,0,0,0,0,0	0	0	0,0	1	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,0	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,0	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,1	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,1	0	0,0	0,0	0
0,0,0,0,0,0,0,0	1	1	1,0	0	0,0	0,1	0
0,0,0,0,0,0,0,0	0	0	0,0	0	0,0	0,0	0

The register file contains an array of 4 registers. The RTM produces two outputs, called abus and bbus; these are readouts of the two registers that are selected by the 2-bit control inputs sreg1 and sreg2. At each clock tick, the register file updates its state; the destination register (selected by the 2-bit dreg control input) discards its old state and updates with the new state. This is either the data input, or the sum of the two selected registers, and the choice is determined by another control input add. The behaviour of the RTM circuit at the register transfer level is equivalent to the following:

```
reg[dreg] :=
  if add=0
    then data
    else reg[sreg1] + reg[sreg2]
abus = reg[sreg1]
bbus = reg[sreg2]
```

There are eight data inputs that comprise a system input bus. The control add bit is 1 if there will be an add, and 0 otherwise. The register load bit is 1 when the register loads. The destination register has two address bits. The register clear bit is 1 if the register is cleared, 0 otherwise. There are two source registers, each with two bit addresses. And the adder takes a carry-in signal.

The RTM circuit has several properties that are characteristic of large circuits, making it a useful tool for evaluating simulation algorithms. In particular, for typical inputs, it is common that many of the internal signals do not change, and for some inputs many of the internal signals are ignored — these are called "don't care" signals. (In contrast, note that a circuit like an adder or signal processor exhibits essentially the same behaviour regardless of the input values.) To exploit the varied behaviour of the RTM, we simulate it with several different sets of input data (these are called programs, as the RTM is a programmable circuit and its inputs comprise the program).

This first program loads a 1 in register zero (twice), a 1 in register one (twice), and then adds the contents of registers zero and one, placing the result in register two. The register read out at the end is 1,1,2,0.

In this second program, a 1 is loaded into register zero, and another in register one (twice). Then the contents of registers

TABLE II
Program B simulated by the RTM

data	add	load	dreg	clr	sreg1	sreg0	ci
0,0,0,0,0,0,0,0	0	0	0,0	1	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,0	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,1	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,1	0	0,0	0,0	0
0,0,0,0,0,0,0,0	1	1	1,0	0	0,0	0,1	0
0,0,0,0,0,0,0,0	1	1	1,1	0	0,1	1,0	0
0,0,0,0,0,0,0,0	0	0	0,0	0	0,0	0,0	0

TABLE III
Program C simulated by the RTM

data	add	load	dreg	clr	sreg1	sreg0	ci
0,0,0,0,0,0,0,0	0	0	0,0	1	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,0	0	0,0	0,0	0
0,0,0,0,0,0,0,1	0	1	0,1	0	0,0	0,0	0
0,0,0,0,0,0,0,0	1	1	1,0	0	0,0	0,1	0
0,0,0,0,0,0,0,0	1	1	1,1	0	0,1	1,0	0
0,0,0,0,0,0,0,0	1	1	0,0	0	1,1	1,0	0
0,0,0,0,0,0,0,0	0	0	0,0	0	0,0	0,0	0

zero and one are added, placing the result in register 2. And then the contents of registers one and two are added, placing the result in register 3. The final register readout is 1,1,2,3.

In the third program, registers zero and one are loaded with 1. The contents of register zero and register one are then added, placing the result in register two. The contents of register one and register two are then added, placing the result in register three. Finally, the contents of registers two and three are added, placing the result in register zero. The register readout at the end is 5,1,2,3.

IV. Synchronous circuit simulation algorithms

In this paper we consider synchronous circuits with a single phase clock that defines points in time called clock ticks. In such a circuit, all state is contained in flip flops, and all flip flops perform a state change simultaneously at a clock tick. There is no feedback in combinational logic gates. This implies that the logic gates can be organised into equivalence classes, where class i (which we call "path depth i") contains a set of logic gates whose inputs are all valid when i gate delays have elapsed after the last clock tick.

A synchronous simulation has an outer loop that iterates over clock cycles. Within this loop, the simulator (1) obtains the values of the input signals for this cycle; (2) calculates all the logic signals, and (3) updates the flip flops. The second step—logic calculation—consists of an iteration over path depths. This must be a sequential loop, as each path depth has data dependencies on the earlier path depths. Within a path depth, there is an iteration over all the components at that path depth, but this can be a parallel iteration because there can be no data dependencies among the components. In realistic systems, the critical path depth is relatively small but the number of components at each path depth can be quite high. Thus the large loops are potentially parallel, and the inherently sequential loops have relatively few iterations. The algorithm has the following structure:

```
cycle := 0
while (more input)
  read all input signals
  for pd := 0 to cpd
    for each c in gates(pd) { simulate (c) }
  for each c in flipflops { c := input(c) }
```

A. Circuit representation

There are many ways to represent a circuit netlist as a data structure. One obvious method is to use a graph which is isomorphic to the circuit, where components are nodes and wires are arcs. However, the algorithms considered in this paper take a different approach, based on flat arrays of integers. The reason for this is that the algorithms use efficient array iterations; we want the representation to be closer to the capabilities of the machines that will run the simulations rather than making it closer to the higher level programming languages.

There are several pieces of information needed for each component, including the component type, the number of inputs and their sources, and the number of outputs and the signals they define. These pieces of information are represented as a contiguous block of elements in a large *component* array. This approach makes the algorithms lower level than they would be in a graph representation, but the data structure can be traversed efficiently on both sequential CPUs and data parallel GPUs.

This array is accessed by arrays of pointers: one each for input and output records and one for each path depth in the circuit being simulated. For each circuit copy, there is an input record, followed by circuit components sorted by path depth, which are followed by an output record. The input record has access to its own copy of the input, and the output record writes the output values of that circuit copy into an output array containing output vectors for the entire simulation.

Each of the component records contains these general fields: the first one has housekeeping information such as the number of inputs and outputs, the second is the input field, and so contains pointers to various outputs of other components with earlier path depths, and the third is the output field, containing the circuit outputs. All of these fields are accessed by using the component pointer to the record plus a defined constant.

B. The sequential brute force algorithm

The sequential brute force algorithm follows exactly the structure of the basic synchronous simulation algorithm shown in the beginning of this section. This algorithm is called "brute force" because it recalculates every signal value on every clock cycle. As we show below, there are various optimisations that can avoid some of that work.

The outer loop of the algorithm iterates over clock cycles. For our experiments, the simulations run for 15,000 clock cycles. The outer loop reads the input signal values for the current clock cycle, and then uses an inner loop to calculate the signal values in order of path depth.

To interface the outer loop with the actual circuit simulation, we treat each input port as a component that produces an output (i.e. an inport is a signal source) and each output port as a component that receives an input (i.e. an outport is a signal sink). The outer loop reads the input signal values and deposits them into the inport components. Once this is done, for each circuit path depth there is a loop over the component array pointers for that path depth, which simulates those circuits. After all the path depths have been simulated, the outer loop body finishes the clock cycle by fetching the output signal values from the outport components, printing them, and finally updating the flip flops.

C. The parallel brute force algorithm

Most of the loops in the circuit simulation algorithm are inherently sequential because of data dependencies. However, for each path depth there is a loop over the components at that depth, and each of those loops is potentially parallel because the components are independent of each other.

Therefore one method for speeding up the sequential brute force algorithm is to execute the path depth iterations in parallel. This is particularly effective using GPU parallelism, because the GPU programming model supports data parallel iterations over arrays, and this is exactly the structure of the path depth loops.

We have implemented this approach using the CUDA framework with C, running on a GPU.

D. Quiescent signals: the event queue algorithm

If the stable inputs to a component during a clock cycle have the same values that they did on the previous clock cycle, then the outputs of the component will also have the same values as before. Furthermore, it is quite possible for the output of a component to remain unchanged during a clock cycle, even if one of its inputs has changed. If an output signal from a component is unchanged, that means the inputs to one or more components at a later path depth will also be unchanged. Large chains of unchanging signals may exist in the circuit, resulting in regions of the circuit that are quiescent.

The brute force algorithms recalculate every signal on every clock cycle, even if there is no possibility that the signal has changed. An effective optimisation is to record the output values of each component; in the next clock cycle, if none of the inputs to the component have changed, we can simply reuse the previous output value. This can significantly reduce the number of components that have to be simulated.

The standard way to implement this idea is to introduce an event queue [12]. At the beginning of the clock cycle, the event queue is initialised to empty. The source signals at path depth 0 are all checked (these are the circuit inports and the flip flop outputs). For any source signal whose value has changed, all the components that receive the signal as an input are inserted into the event queue. The event queue consists of a set of components at each path depth; whenever a component is inserted into the queue, it must be inserted into the set at the path depth which the component actually has. The main inner loop of the simulation algorithm processes the event queue by repeatedly taking the next component, simulating it, and performing any insertions required if the component has changed an output. For each component with inputs that have changed, the outputs are recalculated, and each component receiving those outputs is placed in the queue. No component with unchanged input values appears in the queue, which is where this algorithm gains its efficiency.

The event queue algorithm is straightforward to implement sequentially using a priority queue data structure.

E. Lock free quiescence: the marking algorithm

The event queue algorithm works well for exploiting quiescence in the circuit as long as the simulator is run on a sequential computer. However, there are significant difficulties with this algorithm on a data parallel host. One problem is that the priority queue is a central data structure that becomes a hot spot; since many threads may need to insert or delete a component at the same time, the basic queue operations must be performed with mutual exclusion, which becomes an increasingly serious problem as the number of parallel threads grows. A second, more general, problem is that the clean iterations across arrays that allow a GPU to implement the parallel brute force algorithm efficiently turn into irregularly structured loops that perform irregular data accesses.

These problems suggest a different approach: can we implement the essence of the quiescence optimisation while avoiding a central data structure that needs locking, and while retaining the simple efficient data parallel looping structure?

We achieve this goal by introducing the *marking algorithm*. The idea is a compromise between brute force and the event queue. Each component has an extra Boolean field, the “mark”, which is initialised to 0 but set to 1 if it is determined that the component needs to be simulated. The marking algorithm processes each component at each path depth, just like the brute force algorithm. Before simulating a component, however, its mark is checked. If the mark is 0 no further work is done on that component; otherwise the component is simulated, its new outputs are compared with the old ones, and for any output that has changed, the components that receive the signal are marked.

The marking algorithm has the same loop structure as the brute force algorithm, making it a good candidate for data parallel systems. We have also implemented it sequentially in order to assess its overheads, but it would normally not be a good choice for sequential simulators.

F. Lock free quiescence: the parallel marking algorithm

The marking algorithm is inefficient for a sequential implementation, but its regularity makes it well suited for data parallelism. A parallel GPU implementation in CUDA is described below.

The marking algorithm avoids any central data structure (e.g. a priority queue) that would require synchronisation. With care, even the marking can be done without synchronisation. Furthermore, the algorithm consists of regular loops over flat arrays, making it fit well with the capabilities of the data parallel GPU architecture.

All components are simulated by an algorithm that runs over an array of pointers into the component array. The way in which these arrays are handled in CUDA helps the program run efficiently. There is a 'loop' that generates threads with ids less than the number of pointers in the array, and simulates the component after determining its component type (in the case of components that are not input, output or latch components).

Here is the loop for components such as multiplexers or full adders.

```
__global__ void interpret_event_circuit
 (int data[],
  int ptrs[],
  int boundary)
{
int tid = threadIdx.x +
          blockIdx.x * blockDim.x;
while (tid < boundary) {
 int ptr = ptrs[tid];
 if (data[ptr] == 1)
   {
   int comp_type
        = data[ptr + COMPONENT_TYPE];
   switch (comp_type) {
    case 5:
     interpret_event_demux24 (data,ptr);
     break;
    case 9:
     interpret_event_mux18 (data,ptr);
     break;
    case 11:
     interpret_event_full_adder (data,ptr);
     break;
    ...
    };
   };
 tid += blockDim.x * gridDim.x;
 }
}
```

Notice that the component gets simulated only if it is marked. The last statement in the code simulating the component sets the 'mark' field to 0 again, so that the component record is ready for the next cycle.

The algorithm for reading inputs takes the component array, a pointer to an input record in the array, and the array of input vectors containing circuit input vectors. It finds the starting location of the 'downstream' field, which contains pointers to components depending upon the input record's outputs. It looks at each input bit in turn.

Starting at the assignment to `j`, which points to the beginning of the downstream field (note that both the inputs field and the outputs field of an input record have the same size, so adding the number of inputs twice makes sense here), the algorithm finds each downstream pointer and marks its component record if the corresponding input bit has changed.

The reason that `k` starts from `j + 1` is that the input record has a count of the downstream pointers for each input bit in `data[j]`. This allows the algorithm to avoid accessing the data array unnecessarily for finding the end of the subfield for that bit.

The algorithm initialises the data structures, using constants that are generated by the hardware description language.

```
__device__ void event_read_input
   (int data[],
    int componentptr,
    int input_vectors)
{
int number_of_inputs
 = data [componentptr +
         IO_COMPONENT_COUNT];
int buffer_ptr
 = data [componentptr +
         IO_COMPONENT_BUFFER_PTR];
int i, k, in_data;
int output_location
 = componentptr +
   IO_COMPONENT_INPUTS_START +
   number_of_inputs;
```

The heart of the parallel marking algorithm is given below.

```
int j = output_location + number_of_inputs;
for (i = 0; i < number_of_inputs; i++)
    {
    in_data = input_vectors [buffer_ptr + i];
    if (in_data != data[output_location + i])
 {
       for (k = j+1; k < j+1 + data[j]; k++)
           data[data[k]] = 1;
       j = j+1 + data[j];
       data[output_location + i] = in_data;
      };
    }
data[componentptr +
     IO_COMPONENT_BUFFER_PTR]
 = buffer_ptr + number_of_inputs;
}
```

It should be clear that with some components using buses, such as multiplexers, the downstream components will tend to be the same for each bit of input. Others, such as the demultiplexer in this circuit, address separate components. Thus there are some redundant marks which can be avoided for a common special case if all of the downstream references are compressed into one, and that was done by our algorithm but isn't expressed here.

G. *Ignoring don't-care signals: the lazy algorithm*

We have considered algorithms that avoid spending time recalculating signals in parts of the circuit that are quiescent. Another way to reduce the work is to observe that some signals will be ignored, and it doesn't matter whether the simulator has the correct value for them. Such signals are traditionally called *don't care* values. An example is a multiplexer that has two data inputs x and y; it will output the value of one of these inputs (depending on a control input c) but it will ignore the value of the other input, which is a "don't care" value. We call simulators that exploit this phenomenon *lazy* (lazy evaluation is an optimisation that avoids doing work that will never be needed).

Computer architecture engineers can make good guesses as to where their machines spend time doing work that may sometimes not be needed. For example, if we treat the inputs to multiplexers as buses, then multiplexers will not need one of their arguments (which one depends on the value of the select bit). If the components generating the values on the unneeded bus are not supplying these values to other components as well, and if the select line of the mux in question has a small path depth, then the circuit may be able to afford not to simulate those components. Multiplexers can play an important role in acting as gatekeepers between subsystems, and so optimisation of their inputs can potentially be worthwhile.

There are two implementation issues for implementing laziness in a circuit simulator: (1) determining which components to check for lazy inputs, and (2) organising the simulation so that the don't care signals for those selected components are not simulated.

In general, the first task—deciding where to consider laziness—is critical because it does introduce overheads, and could be applied almost everywhere. For example, every *and* and *or* gate can induce don't care values, and these are among the most common logic gates used.

Rather than implement a laziness analysis, we decided first to investigate whether it would be worthwhile. To do this, we manually analysed the RTM circuit, selected one component that could save significant time by ignoring it's don't-cares, and adapted the simulator to handle this special case. It turns out that in the RTM circuit, the mux selecting either the adder output or the data input is suitable. We had the circuit simulator software check the control add bit. If it was 0, then the adder's output would not be needed, meaning that a long line of components need not be simulated. The results of this experiment (discussed below) are helpful in assessing whether an automated analysis would be worthwhile. Such an analysis could generate a table of interesting muxes based on the depth of the input path, or other criteria, allowing the designer to select which muxes to optimise. This turns out to work fairly well for the sequential implementations of the Brute Force and Marking algorithms.

The central loop of the sequential lazy algorithm is given below. After the inputs are read, a conditional checks whether the control add signal is 1, then simulates the components of path depth 1 to 10 if it is. The constant `INITIO_VECTOR_LEN` is the number of input vectors containing the program to be simulated by the RTM. The constants `INITDEPTH_1_LEN` up to `INITDEPTH_11_LEN` represent the number of components at that path depth in the original circuit. When one of these constants is multiplied by `N`, the number of copies of the circuit, the result is the number of components at that path depth.

```
for ( cycles=0;
      cycles < INITIO_VECTOR_LEN;
      cycles=cycles + 1) {
read_inputs ( data,
input_ptrs,
N,
input_vectors);
if (input_vectors
     [data[IO_COMPONENT_BUFFER_PTR] -
      INPUT_RECORD_SIZE + CTL_ADD]
     == 1)
 {
  interpret_circuit ( data,
  depth_1_ptrs,
  INITDEPTH_1_LEN * N);
  ...
  interpret_circuit ( data,
  depth_10_ptrs,
  INITDEPTH_10_LEN * N);
 };
interpret_circuit ( data,
depth_11_ptrs,
INITDEPTH_11_LEN * N);
write_outputs ( data,
output_ptrs,
N,
output_vectors);
interpret_latches ( data,
latch_ptrs,
INITLATCH_LEN * N);
};
```

V. Results

To assess the performance of the simulation algorithms, we have implemented each algorithm in sequential C (for a CPU) or parallel C+CUDA (for a GPU), and measured them as they simulate the RTM circuit. Furthermore, we measure performance separately for the three different RTM input programs. The simulations all run for the same number of simulated clock cycles (15,000).

The sequential Event Queue, Brute Force, Marking and Optimised Brute Force and Marking algorithms were simulated on a 2.3GHz Pentium processor. The parallel Brute Force and Marking algorithms simulated the RTM as it executed the three test programs on a GeForce GTX 590 GPGPU with Cuda capability 2.0, 512 cores and 1.22 Ghz clock speed. Each test program was simulated five times. The measurement results are given below.

TABLE IV
SEQUENTIAL TIMINGS FOR MARKING AND BRUTE FORCE ALGORITHMS (N=1000, 5000 AND 9000). BF = BRUTE FORCE ALGORITHM; EQ = EVENT QUEUE ALGORITHM; M = MARKING ALGORITHM; L = LAZY ALGORITHM; LM = ALGORITHM WITH BOTH MARKING AND LAZINESS.

		Avg	St Dev	Avg	St Dev	Avg	St Dev
		1000	1000	5000	5000	9000	9000
BF	A	170.00	0.00	2398.00	209.95	4374.00	409.77
	B	171.80	0.45	2320.40	6.02	4191.40	30.17
	C	170.60	0.89	2302.20	14.91	4203.80	34.22
EQ	A	167.00	0.71	1300.80	5.07	2342.40	12.46
	B	210.00	1.73	1701.40	6.11	3063.60	5.03
	C	255.20	0.84	2116.60	12.66	3797.20	10.28
M	A	196.80	0.84	1454.80	1.30	2585.40	8.35
	B	216.20	0.45	1672.80	5.02	3020.00	10.93
	C	238.20	0.45	1947.40	5.94	3519.20	10.31
L	A	107.20	0.45	1103.80	4.32	1998.00	13.64
	B	118.20	0.45	1303.00	9.03	2362.00	15.35
	C	129.00	0.71	1498.20	4.76	2721.60	12.78
LM	A	142.00	1.22	1109.40	5.86	2006.20	13.48
	B	169.00	0.00	1378.00	4.06	2488.40	10.60
	C	190.60	1.34	1597.00	10.44	2872.60	8.96

Each test program was simulated five times. Again, the averages and standard deviations appear in the tables.

A. Sequential optimisations

Table IV shows the performance of the sequential algorithms.

Several observations can be made from the data:

- The quiescence optimisation using the event queue gives a large improvement in performance compared with the brute force algorithm. This would not hold for all circuits—indeed, it is possible to construct an artificial circuit that would make the optimisation increase the time—but the RTM has the same structure as many realistic circuits.
- The quiescence optimisation using the marking approach saves time compared with the brute force algorithm, but is less efficient than the event queue. It was expected that sequential marking would be slower than sequential event queue, because the marking algorithm still has to execute a statement or two for every component, even the quiescent ones. It is encouraging—and somewhat surprising—that sequential marking outperformed sequential brute force. This means that the work saved exceeded the overheads introduced.
- The laziness optimisation was also very effective.
- The combination of marking and laziness did not do as well as laziness by itself. However, this may be an artificial artifact of the experimental setup: if the RTM is extended to a full CPU circuit, there will be large parts of the circuit that are quiescent and *different* parts that are don't cares, but this is not the case with RTM. Further research is needed to assess the combination of these two optimisations.

TABLE V
TIMINGS FOR MARKING AND BRUTE FORCE ON A GPU (N=1000 TO N=3000)

		Avg	St Dev	Avg	St Dev	Avg	St Dev
		1000	1000	2000	2000	3000	3000
M	A	76.32	0.00	86.26	0.01	107.33	1.18
	B	86.42	0.01	97.14	0.01	119.96	0.02
	C	97.06	0.00	108.79	1.21	132.48	0.01
BF	A	89.69	0.16	101.52	0.17	122.74	0.20
	B	89.60	0.16	101.47	0.16	122.66	0.23
	C	89.57	0.16	101.45	0.17	122.68	0.22

TABLE VI
TIMINGS FOR MARKING AND BRUTE FORCE ON A GPU (N=4000 TO N=6000)

		Avg	St Dev	Avg	St Dev	Avg	St Dev
		4000	4000	5000	5000	6000	6000
M	A	130.52	1.13	168.01	0.01	194.65	0.02
	B	145.61	0.01	190.39	0.01	218.37	0.01
	C	160.57	0.01	211.93	0.01	241.05	0.01
BF	A	146.59	0.35	189.73	0.47	216.33	0.50
	B	146.59	0.36	189.88	0.49	217.33	1.61
	C	146.65	0.35	190.08	0.49	216.47	0.51

TABLE VII
TIMINGS FOR MARKING AND BRUTE FORCE ON A GPU (N=7000 TO N=9000)

		Avg	St Dev	Avg	St Dev	Avg	St Dev
		7000	7000	8000	8000	9000	9000
M	A	227.09	0.01	259.23	0.01	327.82	0.01
	B	253.64	0.01	289.37	0.01	367.58	0.03
	C	278.56	0.02	318.16	0.02	406.72	0.02
BF	A	251.20	0.53	283.17	0.73	363.43	1.52
	B	251.73	1.31	283.31	0.71	362.92	0.85
	C	251.43	0.54	283.37	0.69	363.08	0.84

B. Parallel marking vs. brute force

The performance of the parallel marking algorithm is shown in detail in Tables V, VI, and VII.

The experimental results suggest the following conclusions:

- The performance of the marking algorithm is fairly close to that of the parallel brute force algorithm. Sometimes it is a little faster, sometimes a little slower, and this depends on the behaviour of the simulated circuit as it reacts to its inputs.
- It would be useful to find out whether we can take a circuit and characterise it in order to decide whether to apply the marking optimisation to the parallel simulation.
- A particularly encouraging point is that, in some cases, the marking optimisation gives a further improvement beyond the significant speedup attained from the parallel host.

TABLE VIII
SPEEDUP FACTOR FOR MARKING AND BRUTE FORCE ALGORITHMS WHEN CPU COMPARED TO GPU (N=1000,5000 AND 9000)

		1000	5000	9000
M	A	2.58	8.66	7.89
	B	2.50	8.79	8.22
	C	2.45	9.19	8.65
BF	A	1.90	12.64	12.04
	B	1.92	12.22	11.55
	C	1.90	12.11	11.58

C. Parallel vs. sequential performance

The table compares the parallel with sequential performance for both the brute force algorithm and the quiescence optimisation using marking. In both cases, the GPU gives a significant speedup over sequential performance.

After writing a variety of CUDA programs to support circuit simulation, using atomic mutual exclusion and thread barriers, and trying the use of shared memory as a cache (with similar conclusions to Langdon [17]), we have come to a conclusion that seems to be common amongst CUDA programmers—namely, rewrite the code until done idiomatically in CUDA. To us, this means 'avoid the fancy stuff if possible', as it has a nasty effect on code efficiency. During our experiments, we found that the Brute Force algorithm was the one to beat. It seems likely that this is because the loops running over the component records are easily done using threads and blocks in CUDA, and barriers are enforced by using kernels in sequence rather than explicitly (where they incur a noticeable runtime cost).

It is surprising that the Marking algorithm can compete, given the marking overheads. However, for some inputs, it can. The marking algorithm loses some of the benefits of the event queue algorithm, because the event queue algorithm can avoid looking at whole groups of components, while the Marking algorithm has to look at each component record, even if it appears in a large group of components, none of which are simulated. But because it is data parallel, the Marking algorithm can be executed on a GPU quickly enough to discount expensive overheads some of the time.

VI. CONCLUSION

We have implemented several related simulation algorithms for synchronous digital circuits, and evaluated their performance as they simulate an RTM circuit. The RTM is simple but realistic, and in fact is the core of a simple processor. As the RTM is a programmable device, its behaviour depends on its inputs, and we have simulated the RTM as it runs three different programs.

The simulation algorithms are all related to a basic brute force sequential algorithm that directly embodies the semantics of the synchronous circuit model. We have modified the basic algorithm in several ways: by introducing two different optimisations (individually and together), and by targeting a data parallel GPU architecture.

The experimental results suggest the following conclusions:

- There is no single best simulation algorithm; there are optimisations that save time but also introduce overheads, and the characteristics of the circuit need to be considered in order to decide which optimisations to apply.
- On a sequential host, optimisations that lead to complex data structures and irregular loops are effective.
- On a data parallel GPU host, optimisations that avoid locking and that keep regular loop structures are effective.

This work indicates several fruitful lines for future work, including: (1) further optimisations related to the semantics of the circuit model; (2) further optimisations aiming to improve low level performance on the GPU; (3) static analyses of the circuit to apply optimisations where they are likely to be most effective.

REFERENCES

[1] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General Purpose GPU Programming*. Addison Wesley, 2010.

[2] S. D. Johnson, "Applicative programming and digital design," in *11th Annual ACM Symposium on Principles of Programming Languages*. ACM, 1984, pp. 218–227.

[3] J. O'Donnell, "Hardware description with recursion equations," in *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*. Amsterdam: North-Holland, April 1987, pp. 363–382.

[4] ——, "Overview of Hydra: A concurrent language for synchronous digital circuit design," in *Proceedings 16th International Parallel & Distributed Processing Symposium*. IEEE Computer Society, April 2002, p. 234 (abstract), workshop on Parallel and Distribued Scientific and Engineering Computing with Applications—PDSECA.

[5] J. Wakerly, *Digital Design: Principles and Practices*, 4th ed. Pearson, 2006.

[6] Z. Wang and P. Maurer, "Lecsim: A levelized event-driven compiled logic simulator," in *27th ACM//IEEE Design Automation Conference*, vol. ACM Transactions on Design Automation of Electronic Systems, 1990, pp. 491–496.

[7] Y. Liu and J. Hu, "Gpu-based parallelization for fast circuit optimization," in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 943–946.

[8] S. Meyer, "A data structure for circuit netlists," in *25th ACM/IEEE Design Automation Conference*, 1988, pp. 613–616.

[9] B. Wan, B. P. Hu, L. Zhou, and C. J. R. Shi, "Mcast: an abstract-syntax-tree based model compiler for circuit simulation."

[10] J. L. Tripp, "Trident: From high level language to hardware circuitry," *Computer*, 2007.

[11] D. Chatterjee, A. DeOrio, and V. Bertacco, "Gcs: High-performance gate-level simulation with gp-gpus," in *Design, Automation and Test in Europe (DATE)*, 2009, pp. 1332–1337.

[12] ——, "Event-driven gate-level simulation with gp-gpus," in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 557–562.

[13] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with gpu computing," *ACM Trans. Design Automation for Electronic Systems*, vol. 16, no. 3, June 2011.

[14] K. Gulati, J. Croix, S. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 403–408.

[15] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Ninth International Symposium on Parallel and Distributed Computing*, 2010, pp. 71–78.

[16] L. Suresh, N. Rameshan, M. S. Gaur, M. Zwolinski, and V. Laxmi, "Acceleration of functional validation using gpgpu," in *6th IEEE International Symposium on Electronic Design, Test and Application (DELTA)*, 2011, pp. 211–216.

[17] W. Langdon, "Debugging CUDA," in *Proc.13th annual conference companion on genetic and evolutionary computation (GECCO'11)*, 12–16 July 2011, pp. 415–422, doi¿10.1145/2001858.2002028.