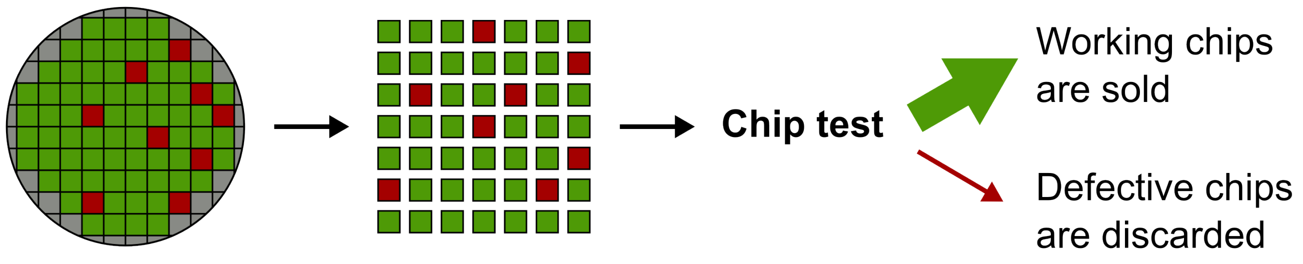
18-645 Term Project – Final Report
December 4, 2012
Matthew Beckler and George Davis

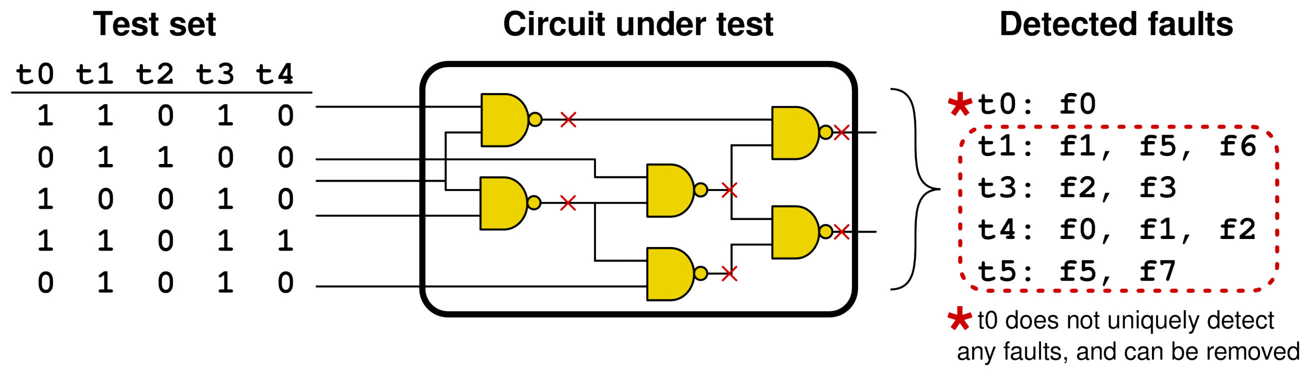
Accelerated and Enhanced Circuit Fault Simulation Using CUDA

1. Problem Motivation and Overview

The manufacture of integrated circuits is an imperfect process. Following the fabrication process, each die on each wafer must be packaged and tested to confirm that it behaves as expected and can be sold, or discarded if it behaves errantly. While hundreds of die can be fabricated in parallel on the same silicon wafer, the testing process requires multi-million dollar pieces of test equipment that can only test a single die at a time. This means that testing becomes a rather serial process, making test an expensive part of the total manufacturing process. Reducing the amount of time spent on IC test is an important problem in this field.



The most promising way to reduce total test time is to reduce the number of tests applied to each die, while still maintaining a high quality set of tests, that is, detecting the same faults with fewer test vectors. In order to evaluate the quality of a test set (the number of faults detected by the tests), a process called *fault simulation* is performed. Fault simulation consists of performing a logical evaluation of the boolean logic values that propagate through a circuit, in the presence of a number of faults in the circuit design that can affect the propagated values. Fault simulation is a very time-consuming process, and finding a methodology to reduce the wall-clock time required would allow for higher-quality test sets to be produced for the same effort, saving money in the semiconductor fab at test time.



Additionally, as part of Matthew's PhD research, alternative fault models (abstractions of the many ways in which a chip can experience a failure) that more closely match real-world failures are being investigated. The new fault model called the TRAnsition-X (TRAX) model matches realistic failures better than existing models, but requires a small amount more processing time for fault simulation due to to enhanced fault activation and propagation conditions. This increase in complexity further motivates the exploration and development of high-speed fault simulation environments. Furthermore, the applications of the TRAX model is for diagnosis using fault dictionaries, which requires full fault simulation without dropping, a very common algorithmic optimization utilized by commercial fault simulators, which will be explained in the next section. The TRAX model, an extension of the conventional transition fault (TF) model, is used in the context of 2-

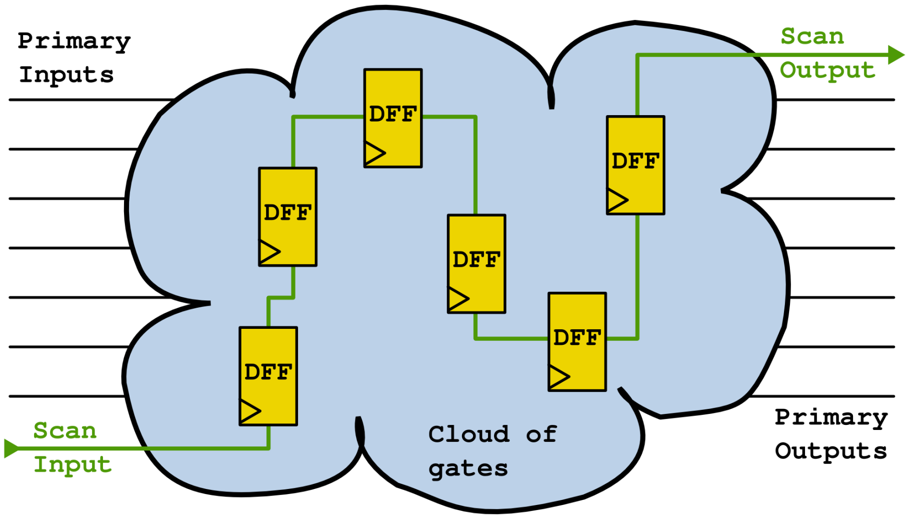
vector signal value transitions. This means that our fault simulation will not be simply on a single set of values (one per signal line) but rather a pair of values (two per signal line), one for the first test vector and one for the second test vector.

2. Related Work

There have been some papers in this research area before, notably [1-3], which have achieved speedups in the range of 20-40x improvement over other CPU and GPU implementations. However, the previous implementations are not fully suitable as a starting point for our work, due to the more complex fault model employed here. First, and most importantly, almost all existing fault simulators perform what is known as "fault dropping". Given that most fault simulators are used only to determine the "fault coverage" of a given test set (i.e., how many modeled faults are detected), once a fault is detected for the first time it does not need to be considered further. This is called fault dropping, and while it is very common in the prior work, it is not an option for this project, as we require the full circuit response in the presence of each fault, for each test in the test set. Since we must simulate all fault-test pairings, this influences our choice of implementation.

While existing solutions may not necessarily be directly applicable, we have been able to borrow some useful and interesting ideas from the literature. For example, some implementations enforced restrictions on the structure and composition of the circuit netlists used with their implementations, and we have chosen some of these same restrictions to apply to our implementation. For example, to regularize the memory structures needed to represent each gate in the circuit, some publications restricted their circuits to only containing gates of two or fewer inputs, such as a two-input NAND gate, or an inverter or buffer. We have similarly chosen to also restrict our implementation to circuits with two or fewer inputs.

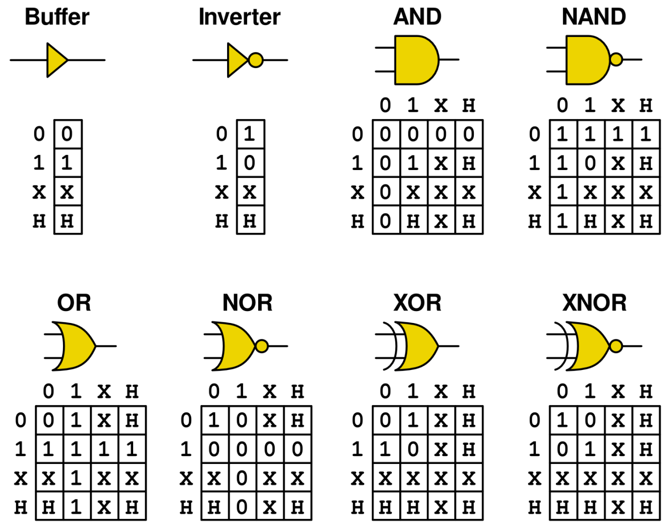
Another restriction we are assuming is a "full-scan" circuit implementation, where all sequential elements such as flip-flops are replaced with a "scan" equivalent. All scan flip-flops in a circuit are connected together in a long chain, which permits the stored value to be both controllable and observable, by loading/unloading the stored values serially along the scan chain. This effectively converts a sequential circuit into a purely-combinational circuit for the purposes of test, making the test generation and application much easier. Adding scan chains to a circuit is an extremely common design-for-test strategy, and is widely used in industry.



By having a purely-combinational circuit netlist, we can treat the circuit graph as a directed-acyclic graph (DAG), which enables some computation-saving optimizations. For example, we can topologically sort the nodes (gates) in the graph (circuit), enabling a one-pass, in-order gate evaluation strategy. More details on this and other optimizations are presented in the GPU implementation section.

3. Fault simulation specifics

For the purposes of this fault model and implementation, we are using a four-valued boolean logic for our circuit simulations, consisting of the traditional 0 and 1 values, with the addition of an "unknown value" X, and a "hazard" value H. The hazard value arises when a gate output may potentially experience an output glitch (0-1-0 transition). The truth tables for the eight primitive gates need to be extended to handle this four-valued logic, as shown below, along with an example of how a hazard H value may arise in the situation of inputs of 10 and 10 to an AND gate, where the actual behavior depends on the exact arrival times of the inputs' transitions, which is simply marked as as hazard (the potential for a glitch).


Buffer
Inverter
AND
NAND
OR
NOR
XOR
XNOR

Buffer

0	0
1	1
X	X
H	H

Inverter

0	1
1	0
X	X
H	H

AND

	0	1	X	H
0	0	0	0	0
1	0	1	X	H
X	0	X	X	X
H	0	H	X	H

NAND

	0	1	X	H
0	1	1	1	1
1	1	0	X	H
X	1	X	X	X
H	1	H	X	H

OR

	0	1	X	H
0	0	1	X	H
1	1	1	1	1
X	X	1	X	X
H	H	1	X	H

NOR

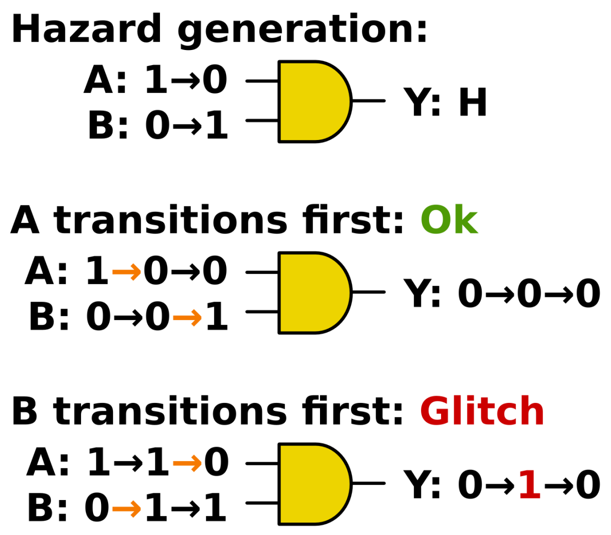
	0	1	X	H
0	1	0	X	H
1	0	0	0	0
X	X	0	X	X
H	H	0	X	H

XOR

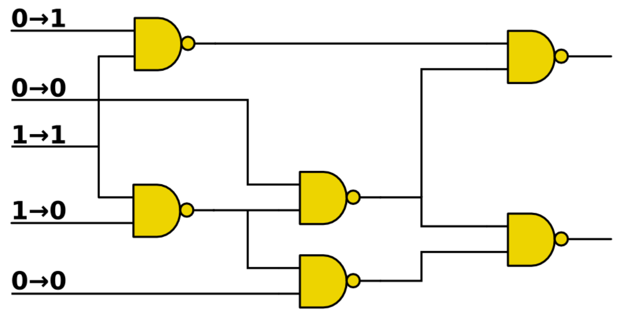
	0	1	X	H
0	0	1	X	H
1	1	0	X	H
X	X	X	X	X
H	H	H	X	H

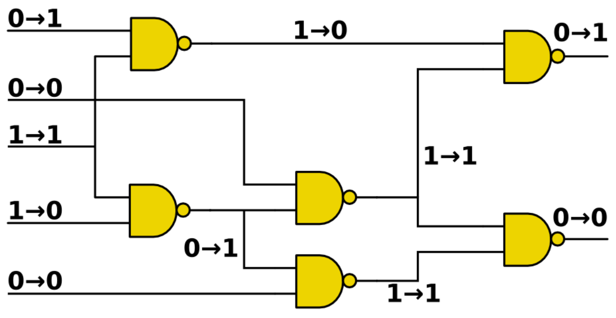
XNOR

	0	1	X	H
0	1	0	X	H
1	0	1	X	H
X	X	X	X	X
H	H	H	X	H

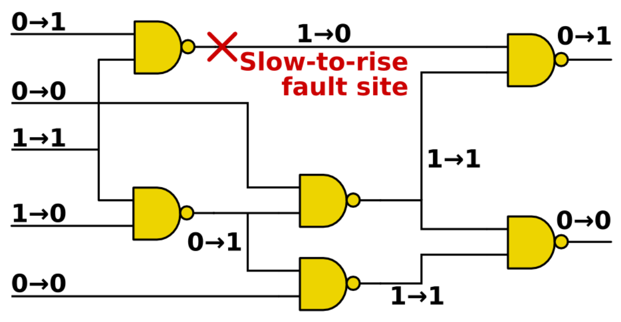


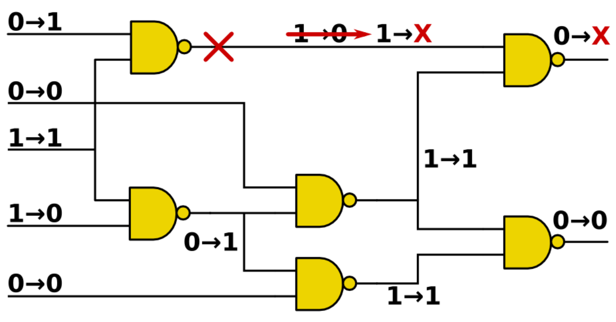
As previously mentioned, we are using a two-vector fault simulation due to the transition-fault nature of the TRAX fault model. The circuit shown in this section is called c17, and consists of six NAND gates with five inputs and two outputs. We start by applying the input transitions to the inputs of the circuit as shown in the figure below on the left. The circuit simulation process then analyzes each gate in turn to determine the output values, producing the fault-free circuit values shown below on the left.





The above method produces the fault-free circuit values (state) for the applied test vector pair. The “fault” aspect of fault simulation is now introduced. Suppose that the current fault site is as indicated below on the left, and it is of the type “slow to rise”, meaning that if the gate output experiences a 1-to-0 transition, the fault is “activated” and produces an X value on the gate output for the second vector. In the image below on the right size, the activated fault produces an X value, which is propagated to the upper circuit output.

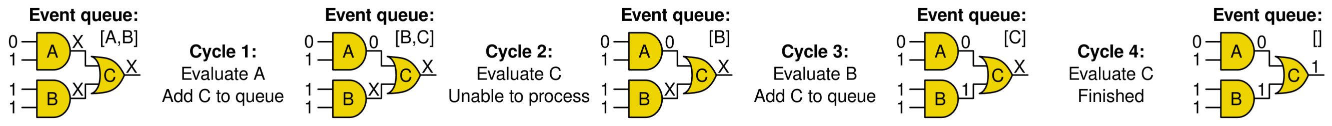




This is the entire fault simulation process for a single test vector pair, for a single fault. We now need to simulate this test vector pair with all the other faults in the circuit, as well as repeating this entire process for all the other test vector pairs in the test set.

4. Reference Implementation

Due to the absence of an existing fault simulator capable of handling the more complex TRAX fault model, we have created a simple, single-core reference implementation ourselves. This has allowed us to think carefully about the input and output data requirements, and we have created a few very simple data types to permit an easy and efficient storage of the circuit netlist and other required data. This single-core reference implementation is an event-based fault simulator. In this common fault simulation technique, changes in the circuit propagate towards the outputs one gate at a time. A queue is maintained to track gates in need of update. After a gate's output changes, all the gates that are fed by that gate are added to the queue and updated in turn. While this is an efficient and straightforward implementation, it is not very applicable to a GPU implementation. An example of evaluating three gates using an event-based system is shown in the figure below. In cycle 2, we assume a random queue for the purposes of making a more interesting example, and show that sometimes a gate on the queue is not able to be evaluated (due to an unspecified input value), and is simply skipped. The gate will be added back to the queue when the other input gate is evaluated.



The reference implementation was written in the Python scripting language, due to its incredible ease of use to create a known-working codebase in the shortest possible amount of time. The reference implementation was run on the GHC cluster machines (same machines as used for the GPU implementation), which have a quad-core Intel Xeon CPU running at 2.67 GHz and 12 GB system memory. While our reference implementation is only a single-core implementation, it would be rather easy to enhance this code to run on as many CPU cores as are available. Again, to emphasize, the reference implementation was developed quickly (under an hour), with primary goals of verifiable correctness and reduced programming time, rather than efficiency or performance.

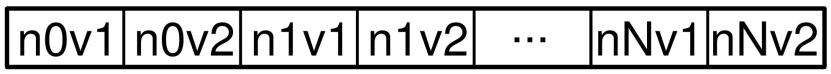
5. GPU Implementation

The GPU implementation has been tricky to implement. We decided to start with the most straightforward GPU implementation possible, and then work to optimize our algorithms, architectures, implementations, and data storage formats to achieve the best performance possible. Our approach has divided the problem into three distinct kernels, utilizing a number of data structures explained below

Data structures

There are a number of important data structures for this problem and its GPU implementation. First, the circuit structure is stored as an array of Gate structures, which tracks the type of the gate (AND, NAND, etc) as well as the two input nets, stored as the net index value. As previously mentioned, the combinational circuit can be thought of as a DAG, which allows us to sort the gates into a topological ordering. The Gate structs are stored in this same order, which permits the fault simulation thread to simply walk the array to evaluate the gates in order, without jumping around in the array.

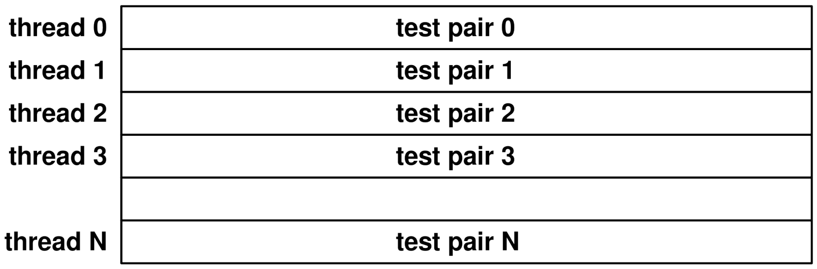
The next data structure is the circuit state, which tracks the values (under test vectors v1 and v2) for all nets in the circuit. The four-valued logic requires two bits for each state value. Originally we simply stored one 2-bit value per byte, however, given the large number of circuit states we must track during execution (see kernel explanations below) we needed to reduce the amount of storage required for each circuit state. Packing four 2-bit values into each byte was a good data storage optimization to implement, that allowed us to attack larger and larger circuits. We assign the nets ID numbers starting with the gate output nets, in the gates' topological order, followed by the circuit primary input values. Each net has its v1 and v2 values adjacent, as shown in the figure below. This permits accessing the v1 value of net_id as **`state[net_id * 2]`** and the v2 value of net_id as **`state[net_id * 2 + 1]`**.



For data such as the input test vectors (series of 0 and 1 bits) we again started with the simplest possible storage format, storing a single 0 or 1 bit value in each byte. Eventually this was upgraded to a compacted "eight bit values per byte" format to enable the code to handle larger circuits.

Kernel 1 – Fault-free fault simulation

The first kernel computes the fault-free circuit states for each test vector pair. This is similar to the task graphically illustrated in the second figure of Section 3 above. This kernel uses a configurable number of threads per block (usually 512), and as many blocks as are necessary to provide one thread per test. The memory for the kernel is shaped and organized as shown in the figure below. Before executing the kernel, however, we first must initialize the circuit's primary input values with the proper values for each test, a task that is performed on the CPU before copying the test pair memory to the GPU.



The kernel itself is very simple, simply computing its **`thread_id`** value and only continuing if it is less than the number of tests. The kernel then calls the **`cuda_fault_sim()`** device function, which performs the actual fault simulation. The reason this takes place in a separate function is that the fault simulation functionality is also used by Kernel 3 (faulty fault simulation), and we can directly re-use this functionality.

Kernel function cuda_fault_sim() - Fault simulation core function

This function is called from both Kernel 1 and Kernel 3, and is the heart of the GPU fault simulation implementation. This is the pseudocode implementation of this function:

```
for each gate g in circuit:
      in1_v1 = state_get(my_test_state, g.in1 * 2)
      in2_v1 = state_get(my_test_state, g.in2 * 2)
      in1_v2 = state_get(my_test_state, g.in1 * 2 + 1)
      in2_v2 = state_get(my_test_state, g.in2 * 2 + 1)
      v1 = gate_eval(in1_v1, in2_v1)
      v2 = gate_eval(in1_v2, in2_v2)
      if hazard_produced(gate_type, in1_v1, in1_v2, in2_v1, in2_v2)
            v2 = H
      state_set(my_test_state, gate_id * 2, v1)
      state_set(my_test_state, gate_id * 2 + 1, v2)
```

Here, we iterate over all gates in the circuit (this is not entirely true, but we will explain the "skip ahead" optimization in Kernel 3). First, the input values to the current gate are loaded. There are four input values, even though we are using 2-input gates, due to the fact that we are applying two input vectors. Using each pair of input values, (in1_v1, in2_v1) and (in1_v2, in2_v2) the gate output values are computed using the **`cuda_gate_eval()`** function. Using the four input values and the gate type, the code checks if a hazard should be produced at the gate output in v2, and if so, sets the value of v2 to H. Finally, we update the state for this test with the (possibly) new values of v1 and v2.

Kernel 2 – Check fault activations

A TRAX fault can only be activated in one of two situations. The first is a traditional gate output transition, either rising or falling (a given TRAX fault is either a "slow to rise" (STR) or a "slow to fall" (STF) fault, and is only activated by the corresponding gate output transition). The second activation situation is when a hazard is present at the gate output (there is no concern about STR vs STF here), either due to hazard-causing inputs at this gate, or having a hazard input value propagate through this gate. In general, each test will activate only a subset of the circuit's faults, and the total runtime can be reduced through only simulating the activated faults. We use Kernel 2 to determine which fault-test pairings need to be fault simulated, using the following data structure, which stores one bit for each fault-test pair, with a zero value indicating that the fault was not activated by this test, and a one value indicated an activation. As with the other data structures, the initial form of this array stored only one bit value per byte, but has since been compacted in order to enable to simulation of larger and larger circuits.

	test 0	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10		test T
fault 0	0	1	0	0	1	1	0	0	1	0	0	···	1
fault 1	1	0	0	1	0	1	1	1	1	0	1	···	0
fault 2	1	1	0	1	0	0	1	0	0	1	0	···	1
fault 3	0	1	0	0	1	0	0	1	0	1	0	···	1
												···	
fault F	1	0	1	1	0	1	1	0	0	0	1	···	0

The kernel invocation has a configurable number of threads per block (512 worked well) and as many thread blocks as are required to provide a thread to each fault. The kernel itself first determines if the thread has any work to do, and if so, enters a loop over each test. For each test, the loop use the gate output values v1 and v2 to determine if either of the TRAX fault activations conditions has occurred:

```
fault_id = blockIdx.x * FAULTS_PER_BLOCK_K2 + threadIdx.x;
gate_id = fault_id / 2;
rising_fault = fault_id % 2; // odd faults are rising, even are falling
if (fault_id < num_faults) {
      for (test_id = 0; test_id < num_tests; test_id++) {
            v1 = state_get(my_test_state, gate_id * 2);
            v2 = state_get(my_test_state, gate_id * 2);
            if ( rising_fault and (v1 == 0) and (v2 == 1)) or
               (!rising_fault and (v1 == 1) and (v2 == 0)) or
               (v2 == H) )
                  fault_activations[fault_id * num_tests + test_id] = 1;
      }
}
```

After Kernel 2 has finished, the resulting matrix of fault activations is further processed to create a list of activating tests for each fault, that is used to guide the launching of the invocations of Kernel 3.

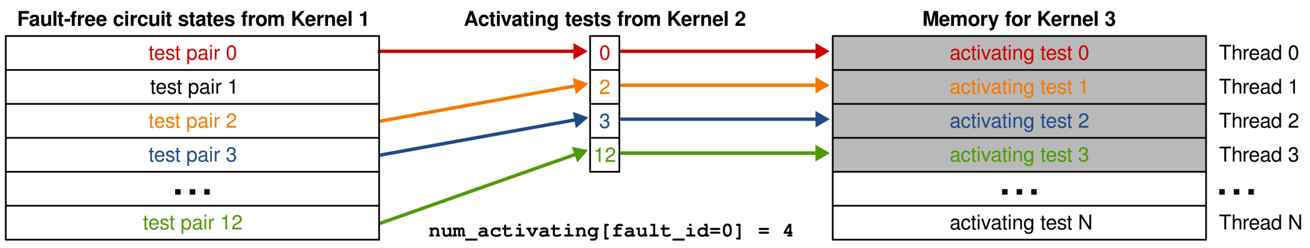
Kernel 3 – Faulty fault simulations

The final aspect of the fault simulation process is to determine how each activated fault affects the circuit values, and if any of the primary outputs are affected by the faulty value. Here, we can utilize a neat trick to speed up the parallel fault simulations that we call the "skip ahead" optimization. Given that the gates are stored in topological order, each thread can simply start at the fault activation point and proceed to process the rest of the gates in order. For fault 0 this is not a speed-up at all, for fault 1 this is a very tiny reduction in work, but for fault N-1, this is a tremendous speed-up. We slightly modify **cuda_fault_sim()** to support starting the fault simulation process at a specified gate, which we use here in Kernel 3 to speed up the fault simulations. Unfortunately, this optimization would introduce branch divergence amongst threads, if different threads in the same block started at a different point in the loop. Initially, we assigned on block to each fault, so that each block of threads would start at the same gate, and stay in lock-step for the entire fault simulation process. However, given that larger circuits will have more than 512 activating tests per fault, it is not possible to maintain this architecture for larger circuits.

To rectify this situation, we decided to structure Kernel 3 so that it would run on only a single fault per kernel invocation. While this would mean that we could have thousands of kernel invocations (and have to pay the kernel launching overhead each time), it permits us to continue to take advantage of the "skip ahead" speed up for each fault.

Another important advantage of this approach is that we do not need to activate a circuit state for each activated fault, which could grow to be a very large amount of memory for larger circuits. Instead, we only need to allocate as many circuit states as the maximum number of activating tests for any fault, since subsequent invocations of Kernel 3 can simply re-use the already-allocated memory. Since each fault can have a different number of fault activations, we also optimize our memory copying calls by only copying the minimum necessary amount of data.

Say, for example, that the fault with the maximum number of activations is activated by eight tests. This means we only need to allocate enough memory for eight circuit states, instead of **`num_tests`** states. Furthermore, let's suppose that fault 0 is only activated by tests 0, 2, 3 and 12. This means that four threads will perform fault simulation, each on a separate circuit state. Using the fault-free circuit states generated by Kernel 1, we copy the corresponding fault-free circuit state for each activating test into the faulty fault simulation memory, as shown in the image below. Since the activating test states are copied into consecutive locations of the Kernel 3 memory array, only the first *n* states (in this example, 4 states) need to be copied into the GPU.



The use of consecutive locations in memory for Kernel 3 enables a very simple kernel, with the pseudocode shown below.

```
test_offset = blockIdx.x * FAULTS_PER_BLOCK_K3 + threadIdx.x;
if (test_offset < num_activations) {
      gate_id = fault_id / 2;
      state_set(states[test_offset], gate_id * 2 + 1, X); // activate fault
      cuda_fault_sim(start = gate_id + 1);
      // fault sim starts with next gate -> "skip ahead" speedup
}
```

6. Analysis of Results

We have performed a series of experiments to provide a comparison between the reference implementation and the GPU implementation. The following circuits are all freely-available, industry-standard benchmark circuits. For each circuit, the critical statistics are listed, along with the wall-clock time (in seconds) for both the reference and GPU implementation. We observed no difference between the reference and GPU in terms of the output data produced, for any circuit; the two implementations produce identical results for the fault simulation.

Circuit	Gates	Tests	Nets	Inputs	Outputs	Ref time	GPU time	Speedup
c17	6	11	11	5	2	0.056	0.071	0.79x
c432	208	125	244	36	7	5.492	0.400	13.73x
b12	1136	418	1262	126	119	145	12.275	11.81x
b14_opt	6878	1713	7153	275	245	11280	1314	8.58x
b14	10681	2322	10956	275	245	97200 (est)	19823	4.9x

For the b14 circuit, the listed time for the reference implementation is only an estimated, due to the very long runtime for this circuit. The estimate was computed by using the first few thousand faults, to estimate the likely total runtime. The reference implementation does not take advantage of the "skip ahead" speed-up, and we believe that this is a fairly-accurate estimate of the runtime.

We observe a good runtime speedup for almost all of the circuits surveyed, with the exception of the very smallest circuit of only six gates, c17. For this circuit, the sub-unity speedup is due to the overhead of the CUDA execution environment and launching kernels on the GPU. For the larger circuits, we see a speedup ranging between 4.9x and 13.73x.

Unfortunately, we observe a troubling downward trend for the speedup values as the circuits grow larger, and are not sure of the cause of this downward trend. We would expect to see an upward trend, where larger circuits help to amortize the CUDA overhead and provide many opportunities for parallelism with a large number of faults and tests. However, this is not the case, and we are continuing to investigate the nature of this trend.

7. Future work

We have a number of ideas for improving the GPU implementation of TRAX fault simulation, and will definitely be pursuing them in the coming weeks. As the TRAX fault model and fault simulation is part of Matthew's thesis research, having a very fast simulator available will be very useful. Here is a list of items we would like to investigate in the future.

- Investigate the use of the constants memory for storing our gate structures. Due to their un-changing nature and the fact that every single thread needs to access the gates data, it seems like a natural fit to put this data into the constants memory.
- Currently the gate evaluation function uses branching to evaluate the complex four-valued logic gates. This is certainly not optimal, and introduces branch divergence. We recently found an idea for using a lookup table approach to gate evaluation, where the gate type and input values are concatenated into an integer address for a lookup table to find the gate output value. This should be simple to implement and reduce the branching evaluation code to a simple array indexing. Additionally, the lookup table should fit nicely in the constants memory as well, especially given the cached nature of that memory.
- As mentioned above, some data storage techniques had to be migrated to a compacted storage technique, where multiple 1 or 2 bit values are stored in a single byte. This reduces the total storage required, but makes each access more complex, usually involving one or more bit shift and bitwise masking operations. It would be good to analyze the actual access patterns for these memories, and determine if functions such as “extract four consecutive values” or “extract both v1 and v2 for a given net” would be useful and faster than individual accesses. Additionally, for smaller circuits or on systems with more available memory, it would be useful to be able to automatically detect the memory size and switch between data storage formats based on the system details.
- There are a number of small optimizations we would like to investigate with regards to array-of-struct vs struct-of-arrays, especially in our gate and test memory storage.

8. References

[1] M. Li and M. Hsiao, “FSimGP2 : An Efficient Fault Simulator with GPGPU,” *Asian Test Symposium*, pp. 15–20, Dec. 2010.

[2] H. Li, D. Xu, Y. Han, K.-T. Cheng, and X. Li, “nGFSIM : A GPU-based fault simulator for 1-to-n detection and its applications,” *IEEE International Test Conference*, pp. 1 –10, Nov. 2010.

[3] M. Kochte, M. Schaal, H.-J. Wunderlich, and C. Zoellin, “Efficient fault simulation on many-core processors,” *Design Automation Conference*, pp. 380–385, June 2010.