GPU Acceleration of Genetic Algorithms for Subset Selection for Partial Fault Tolerance

D. Foster
Electrical and Computer Engineering Department, Kettering University, Flint, MI, USA

Abstract - *As reconfigurable logic devices see increasing use in aerospace and terrestrial applications, fault tolerant techniques are being developed to counter rising susceptibility due to decreasing feature sizes. Applying fault-tolerance to an entire circuit induces unacceptable area and time penalties, thus some techniques trade area for fault tolerance. Area-Constrained Partial Fault Tolerance (ACPFT) is a methodology that explicitly accepts a device's resources as an input and attempts to find a maximally fault-tolerant subset, but determining an optimal partition is still an open problem. While ACPFT originally used heuristics for subset selection, a modification called ACPFT-GA has been developed that uses genetic evolution to provide significantly better fault coverage in many applications. However, its running time is substantially longer than standard ACPFT and may be prohibitive. This paper presents a GPU-accelerated version of ACPFT-GA that has executed over 27 times faster than CPU versions, allowing ACPFT-GA to better scale to larger circuits.*

Keywords: Genetic algorithms, partial fault tolerance, reconfigurable logic, GPU programming

1 Introduction

Two options for a system's processing device are general purpose processors (GPU) and application specific integrated circuits (ASIC). The GPU offers great flexibility but low relative computational power. An ASIC can be designed to provide the greatest processing capability, however this requires a lengthy and very expensive design process, and it is extremely costly for small production runs. A reconfigurable logic device such as a field-programmable gate array (FPGA) offers an attractive third alternative. They provide high levels of computational power like ASICs, yet their ability to be reprogrammed gives flexibility like GPUs. They are off-the-shelf devices and therefore do not have the lead times of ASICs. These features make them common choices in the low production runs of aerospace applications, and they are increasingly used in terrestrial systems. However, they may contain millions of bits to store configurations, and this makes them more susceptible to faults caused by electromagnetic radiation than GPUs and ASICs. Aerospace systems are currently concerned with errors due to single event upsets (SEUs), and as transistor feature sizes continue to shrink, terrestrial systems are also becoming wary[1, 2].

Many applications using reconfigurable logic are not safety critical. A failure can be tolerated by ignoring the error and continuing, such as in video playback. In other cases, the operation can be reattempted, such as retransmitting dropped packets in network communication. However, a reduction of faults would clearly improve the user experience. Since systems implemented with reconfigurable logic invariably leave a portion of the device unused, these extra device resources can be leveraged to provide some level of partial-fault tolerance and reduce the fault rate. The problem of applying partial fault-tolerance can then be formulated as follows. The logic cells contained with the original circuit must be partitioned into a protected subset and non-protected subset such that the fault-coverage is maximized given a set amount of additional logic resources. With current logic devices having hundreds of thousands of logic cells, this presents a gigantic solutions space.

A method of partial fault tolerance called Area-Constrained Partial Fault Tolerance (ACPFT) has been developed that accepts a circuit's available area as an input and finds a maximally fault-tolerance version of the circuit. This initial implementation utilizes difference heuristics to determine a partition, and it generally executes very quickly. A second version called ACPFT – Genetic Algorithm (ACPFT-GA) uses genetic evolution to explore the solution space. It was found to produce significantly more fault-tolerant circuits in expected application spaces, but the running times of ACPFT-GA can be two orders of magnitude larger than ACPFT's. To provide the fault coverage of ACPFT-GA with a more acceptable execution time, the research presented here accelerates ACPFT-GA using NVIDIA CUDA, which is a popular programming extension for running scientific computations directly on massively-parallel graphics processors. This results in an average speed-up of around 17 to 18 times over standard ACPFT-GA with some cases showing speedups of over 27 times.

This paper is organized as follows. Section 2 reviews the key concepts of ACPFT-GA, other efforts to accelerate genetic algorithms with CUDA, and the key considerations when developing with CUDA. Section 3 describes the implementation of ACPFT-GA, the tool chain, and the particulars of casting it as a CUDA-based algorithm. Section 4 presents the experimental results, and Section 5 concludes the paper.

2 Background

2.1 Partial Fault Tolerance

Since this paper focuses on the acceleration of ACPFT-GA and not the introduction of ACPFT-GA itself, readers are referred to [3] and [4] for the justifications of partial fault tolerance in reconfigurable logic. These summarize the advances in partial fault tolerance along with alternatives to ACPFT such as the BYU-LANL partial TMR tool [5], partial error masking [6], selective TMR [7], and Automatic Insertion of Partial TMR [8]

Triple modular redundancy (TMR) remains the standard fault-tolerance method for FPGAs [9]. It can be applied to circuits regardless of the function and of the logic cells used, and it often adds a minimal delay compared to other methods. TMR almost completely protects a circuit against a single fault, although voting logic may still be susceptible. However, it more than triples the circuit's size with a corresponding increase in power use. [10]. These advantages make TMR the most common basis for partial fault-tolerance.

Area-Constrained Partial Fault Tolerance is a technique that uses partial TMR to reduce the circuit area susceptible to faults even if the majority voters are not considered ideal, meaning that they can suffer faults also [3]. Ideal voters is an assumption often used for simplification in other methods because when a large subset of the circuit is being protected, the cross-sectional area of the majority voters is significantly smaller than the tripled area, perhaps by several orders of magnitude, and the rate of faults in the voters is considered to be negligible. This assumption is invalid in a fine-grained approach where the protected area and majority voters have comparable areas. ACPFT was originally designed to use several heuristics and metrics to determine a maximally fault-tolerant partition of a circuit's logic cells.

ACPFT maps well to genetic algorithms since it is similar to the familiar knapsack problem. In knapsack, there is a set of items, each with a weight and a value, and a knapsack that can hold a fixed weight. The optimization problem is to select a subset of items that can be carried in the knapsack with the maximum total value. The additional area of the FPGA relates to the knapsack, and the logic cells with their areas and sensitivities to faults relate to items with weight and value. However, the fault-tolerance problem is more complicated since the additional area required by each logic cell is not a constant value. It is a function of the other cells being protected. Previous research demonstrates that even simple genetic algorithms can create more fault-tolerant partitions than heuristic methods under common conditions, namely that the amount of additional resources available for fault tolerance is less than the size of the unprotected circuit [4].

2.2 Fundamentals of CUDA

CUDA is an extension to several common programming languages, prominently C and C++, which requires an NVIDIA-based graphics processor for execution. NVIDIA GPUs are widely deployed and thus represent a very common computing platform. For easy scalability, NVIDIA cards are designed around a generalized processing unit called a streaming multiprocessor (SM). This allows the performance of CUDA applications to scale based on the number and hardware implementation of the SMs contained on a given card. Details of NVIDIA GPUs can be found at [11].

Since the underlying hardware architecture of a GPU is drastically different than a CPU, algorithms must be crafted using several critical concepts in order to make efficient use of the GPUs computational power [12]. Since GPU-based cards have either a small cache system or none at all, they rely on massive thread parallelism to hide memory access latency by executing a different group of threads when one group block on a memory operation. Therefore, an algorithm must be able to extract sufficient parallelism from a problem to occupy the GPU's thread slots and mask this latency. Second, main memory accesses are efficient only when reading from contiguous memory locations. Data structures and memory accesses should be structured to use this coalesced pattern. Third, a GPU offers several different types of physical memory, such as the large global RAM, small local shared memory, constant memory, per-thread registers, and so on. Careful design should use the most appropriate memory type. Finally, an NVIDIA GPU executes threads in groups called warps. For each clock cycle, all of the threads in a warp or half-warp must either execute the same instruction or do nothing. When threads within a warp execute different code, called divergence, more and more threads will remain idle per clock cycle, and the processing power of the GPU is under-utilized.

With well-crafted algorithms designed with the above considerations, GPUs can potentially execute algorithms significantly faster than CPUs. GPUs can also solve problems using an estimated one tenth to one twentieth of the power required by traditional supercomputing systems [13], thereby reducing costs.

2.3 CUDA as a Platform for Evolutionary Algorithms

CUDA is already established as a popular platform for evolutionary computing. Examples can be found for simple genetic evolution [14] and differential evolution [15]. Ant colony optimization has also been explored [16-18] as well as particle swarm optimization [19-21].

In many evolutionary techniques, the execution time to evaluate the fitness function is a significant fraction of the algorithm's overall time, as is the case with ACPFT. Thus, even though there have been many implementations of evolutionary algorithms using CUDA, and the common operations are becoming well-understood, it is still crucial to explore efficient implementations of new, unique fitness functions and common evolutionary operators that support them.

3 Implementation

3.1 Genetic Algorithm Structure

The solution to the partial fault tolerance partitioning problem is coded as an ordered array of bytes in which each byte corresponds to a specific logic cell in the circuit. The byte's value is '0' if it is in the non-protected partition and '1' if it is in the protected partition. One byte is used per gene in the chromosome instead of one bit since other values are used temporarily during the constraint satisfaction check described later.

The algorithm randomly selects some chromosomes for mutation, choosing those with higher fitness functions proportionally more often. Each gene is examined for random mutation. Mutation results in the binary value being flipped. For crossover, chromosomes are selected in pairs. One gene is randomly selected as the location for single point crossover. Two new chromosomes are created from each pair selected. Mutation is not performed on chromosomes created by crossover.

The fitness function is simply the number of 1's in the chromosome, representing the number of logic cells that are in the protected subset. Previous research has shown that this correlates very highly with the actual amount of fault coverage provided. However, the complexity in ACPFT-GA is that the chromosome must represent a circuit that can fit within the available logic resources. Therefore, the chromosome is processed such that it represents a circuit with ACPFT correctly applied. In this format, a gene is '0' if it is unprotected, '1' if it is a protected and tripled cell, and '2' if the cell is tripled and connected to a majority voter. With this format, the amount of logic resources can be calculated and compared to those available. If the resulting circuit violates the constraints, the chromosome's fitness is set to a value lower than any possible valid chromosome. It is not culled since further evolution may result in a valid chromosome again.

3.2 Tool Chain

Each circuit is represented in a net list in the common EDIF format. ACPFT was written in Perl to accommodate reading and processing this input file and altering it for the partially protected output file. When using heuristics, the partitioning is processed with the Perl script. If the genetic evolution is selected, the ACPFT Perl script parses the EDIF net list outputs a condensed version of the net list in a text file. A C++ program then imports this formatted net list, performs the genetic evolution with or without using a GPU, and outputs the best chromosome. The ACPFT in turn uses this chromosome to partition the circuit and creates the proper modifications of the EDIF net list that can then be implemented on an FPGA. The script also generates a user constraint file to prevent the FPGA tools from removing the redundant logic cells. Currently, the ACPFT tools are run manually, but they are designed such that they could be easily inserted into the standard FPGA design flow and automated.

3.3 CUDA Implementation

The chromosomes are stored in a 2-dimenional array of characters in which each row corresponds to a complete solution. The array contains enough rows to hold all the members of the current generation and those created by mutation and crossover for the next. Once this entire array has been scanned to determine which chromosomes should be carried into the next generation in order of decreasing fitness function, the appropriate chromosomes are copied into a second identical array, and this second array is then the source of members for the next loop.

The GPU's constant memory was used to store many invariants of the net list, such as logic cell type, numbers of destination cells, and lists of destinations cells. It was also used to hold many kernel parameters that are fixed for a given evolution. None of these values are required to be in constant memory for the algorithm to function, so they could be moved to global memory if the circuit is too large for constant memory.

The pseudo code below shows the basic steps that are performed per generation by ACPFT-GA.

1. Generate array of pseudo-random values
2. Randomly select chromosomes for mutation and crossover
3. Create new chromosomes by mutation
4. Create new chromosomes by crossover
5. Calculate fitness function and validate constraints
6. Reorder the array of chromosomes.

CUDA has a random number library called cuRAND that has a CPU-only version. It is used in the CPU implementation so that both the CPU and GPU versions use the same pseudo-random sequences and generate identical output. Also, the cuRAND number generator is effective when generating large batches of values, so it is called once at the beginning of the loop to create all random values for that iteration.

In step 2, some random values are used to select chromosomes for steps 3 and 4. The fitness values of the chromosomes are summed, and each chromosome is assigned a range of values equal to its fitness. Each random number is scaled to the sum of fitness values, and then it is compared to the chromosomes' ranges to determine which is selected. In the GPU version, this operation is performed on the GPU. The kernel is written so that each block is 256 threads, each one converting one floating point random number to an integer corresponding to a chromosome. The number of blocks required is the ceiling of the number of chromosomes needed in the next two steps divided by 256. With the values tested, there were far too few blocks to fully occupy the card, but it is more efficient than transferring data to the CPU for computation.

In step 3, the algorithm selects a chromosome based on the indices from step 2, and it checks each gene for a mutation using random values still remaining from step 1. On the GPU implementation, one block is launched for each chromosome being mutated. The block size is set to 256, with each thread

checking every 256th gene for mutation. This allows the SM of a revision 2.0 GPU to hold 6 blocks, and the number of blocks/chromosomes needed to fully utilize the card is only a few dozen to a couple hundred.

The crossover step is similar to the mutation step, using indices from step 2 and the remaining random values from step 1. A pair of chromosomes is handled by one block, and each block uses 256 threads, again with each thread processing every 256th gene. Twice the number of chromosomes is needed to occupy the GPU than in step 3, but this number was easily reached.

Step 5 is by far the most complex. At this point, a chromosome contains only 1's and 0's representing the protected and non-protected subsets of logic cells. In ACPFT, each protected cell must be tripled, requiring three of that type of logic cell. For each tripled cell, all of the cells that use its output and are still classified as non-protected must be tripled and then combined with a majority voter. This step is performed first and is designed for as many coalesced accesses as possible, although some are unavoidable when examining a cell's destinations.

Once cells have been promoted to tripled and voted, each cell with a voter is examined to see if all destination are tripled or voted. If so, the logic cell can be converted to a tripled cell, increasing the size of the protected subset and actually freeing some resources used for voters. This step also has coalesced and non-coalesced accesses.

After this step, the fitness function is calculated with a reduction from the CUDA thrust library. The sum consists of the sensitivities for all cells that are in the protected subset, ignoring cells that are single or tripled. Next, the constraint condition is checked. Each logic cell adds a count to the logic cells used based on its state. Single cells add one to the like type, tripled cells add three to the like type, and voted cells add three to the like type and one to the type used for voters. These counts are contained in shared memory and require atomic addition instructions to avoid races.

Once all of the logic cells are accounted for, one thread compares the needed resources to the available resources. If the constraints aren't met, the fitness value is adjusted to 1.0, so that the invalid chromosome still has a small chance of being selected in the next generation. Further mutation and crossover may again result in a valid cell.

The CUDA thrust library is used in step 6. The fitness values are sorted and the new chromosome order is determined using these optimized sorting functions. Another thrust function calculates the prefix sum used for the chromosomes ranges. Finally, a kernel uses the new order to copy the best chromosomes from the current chromosome array into the second array, and the pointers to these arrays are swapped in preparation for the next generation.

4 Testing and Results

ACPFT-GA was tested using the *alu4*, *apex2*, and *pdc* circuits from the ACM/SIGDA "Big 20" benchmarks. These circuits were chosen since they show a range of circuit sizes with 597, 1056, and 1328 logic cells respectively. They have also been used in previous research, and data exists for comparisons.

As with previous ACPFT experiments, the performance given different amounts of available resources is accomplished by creating an array of theoretical FPGAs of varying sizes. The number of logic cells in each circuit is used as a "perfect-fit" FPGA. Larger devices are emulated by increasing the number of each resource by a constant multiplier and rounding down. This method created up to 23 theoretical FPGAs, from 10% to 230% in increments of 10%. Between 210% and 230% depending on the test circuit, there were sufficient resource for full TMR, and partial fault-tolerance would no longer be necessary.

The genetic algorithm parameters were selected to match those used in [4]. The test used a set of 4096 chromosomes. For each circuit, the mutation factor was the reciprocal of the number of logic cells. In each generation, the top 256 chromosomes were carried over into the next generation. 1920 were selected using elitism for mutation, and each gene was checked for a mutation. The remaining 1920 chromosomes were generated by crossover. Mutations were not applied to chromosomes created through crossover. Like the previous work, each initial chromosome was initialized to a string of "0"s, representing a fully unprotected circuit. Three more experiments were performed using the output of a heuristic method from previous research to initialize the chromosomes. The fanout method was chosen since it yielded very good results and had a low execution time. For the second experimental setup, all additional resources were supplied to the fanout method, and then the result was refined with the genetic algorithm. In the third and fourth setups, the fanout method was supplied 10% fewer and 20% fewer resources than available respectively. In these cases, there were still some unallocated logic cells when the genetic algorithm was applied.

The test computer used a Core i7 processor at 2.8 GHz, 6 GB of RAM, a GTX 480 graphics card, and CUDA SDK 4.0. ACPFT-GA was run ten times on each simulated FPGA using just a CPU and then using the GPU. Each test was allowed to run for 1000 generations. The times required for mutation, crossing, and calculating the fitness functions and constraint conditions were logged for all runs.

The mutation time, crossover time, fitness function and constraint checking time, and total execution time are shown in the following tables. Times are shown for the CPU-only version, the GPU accelerated version, and the speedup of the GPU version relative to the CPU version. The data for *alu4*, *apex2*, and *pdc* are shown in Table 1, Table 2, and Table 3 respectively for the tests that begin with no initialization, i.e. all available resources are unused and selected only by the genetic algorithm. The data for tests in which 20% of the resources are left unused, and the chromosomes are initialized with the output of ACPFT using fanout are shown in Table 4, Table 5, and Table 6 for *alu4*, *apex2*, and *pdc* respectively. The data for the other two tests are not shown due to space limitations, but the results are very comparable to the 20%

used tests. These tables show results from 10% additional resources to 230% additional resources.

The data shows several patterns. First, the amount of time required for mutation and crossover remains fairly fixed for each circuit over the range of additional resources. This is expected, since the work performed for the mutation and crossover steps depends on the number of genes and the number of chromosomes. From circuit to circuit, the differences in mutation times and crossover times for both versions were smaller than the difference in circuit sizes. This indicates that these steps are communication bound, as is expected.

The fitness and constraints checking execution time shows much more variation. The pattern of this variation is shown in Figure 1 for the *pdc* circuit with no initialization accelerated with the GPU, and it is representative of graphs of other tests.

This graph demonstrates that the amount of time required for the fitness function is very dependent on the amount of additional resources made available. This pattern is logical. To evaluate the fitness function, each cell that is within the protected subset must be examined to triple and vote its output cells, followed by examing voters to see if they can be removed. As the amount of available resources increases, the number of cells within the protected partition increases, and thus the execution time required also increases.

The data shows that the mutation and crossover speedups are comparable between circuits. The tables also demonstrate that the crossover time consumes a few percent of the total time, the mutation time is usually within 10% to 20% of the execution time (with lower percentages as more resources are made available), and the fitness function consumes the majority of the processing time. Therefore, the total speedup depends largely on the speedup of the fitness function. The speedups between circuit is also very similar. All three circuits were implemented with the same time of logical device and had about the same average fanout. Therefore, the amount of work per logic cell in the protected subset was roughly the same for all three circuits.

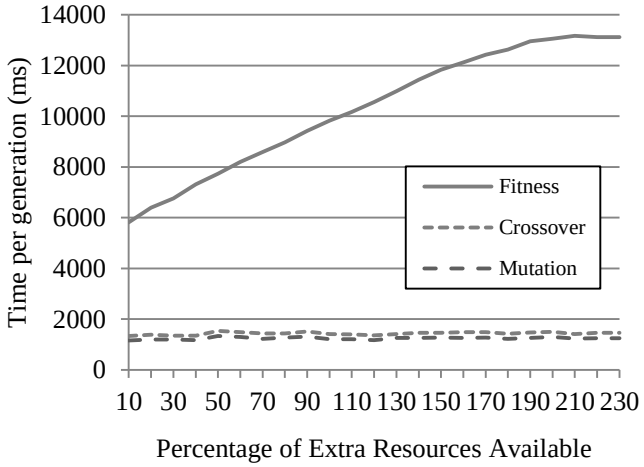

Figure 1 Per Generation Execution Time for the *pdc* Circuit using a GTX 480 and 20% Unused Additional Resources

Table 1 Performance of *alu4* with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	15,874.2 ms	5,874.8 ms	55,103.7 ms	77,068.7 ms
Max. time w/CPU	17,267.5 ms	7,044.1 ms	89,340.1 ms	113,767.3 ms
Ave. time w/CPU	16,641.1 ms	6,478.9 ms	76,733.0 ms	99,968.7 ms
Min. time w/GPU	856.4 ms	123.7 ms	3,011.3 ms	5,081.7 ms
Max. time w/GPU	1,134.8 ms	185.1 ms	7,093.1 ms	19,412.1 ms
Ave. time w/GPU	1,068.7 ms	153.9 ms	5,504.3 ms	12,742.3 ms
Min. Speedup	14.0	32.1	12.4	13.4
Max. Speedup	20.0	54.1	17.3	17.5
Ave. Speedup	15.6	42.5	14.3	15.1

Table 2 Performance of *apex2*with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	25,186.7 ms	7,774.0 ms	108,212.6 ms	144,558.5 ms
Max. time w/CPU	26,760.9 ms	8,917.8 ms	158,688.5 ms	194,473.5 ms
Ave. time w/CPU	27,102.6 ms	8,420.2 ms	142,759.2 ms	179,140.1 ms
Min. time w/GPU	1,158.2 ms	154.6 ms	4,481.6 ms	8,743.7 ms
Max. time w/GPU	1,340.5 ms	232.3 ms	11,750.8 ms	47,051.5 ms
Ave. time w/GPU	1,243.5 ms	194.2 ms	8,850.8 ms	29,703.4 ms
Min. Speedup	19.5	36.8	12.9	14.0
Max. Speedup	22.4	56.4	23.3	24.2
Ave. Speedup	21.8	43.8	17.1	18.2

Table 3 Performance of *pdc* with no initialization

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	31,728.3 ms	8,887.0 ms	131,546.4 ms	172,293.0 ms
Max. time w/CPU	32,522.6 ms	9,804.8 ms	194,660.3 ms	236,821.8 ms
Ave. time w/CPU	32,185.7 ms	9,396.8 ms	174,245.2 ms	215,949.1 ms
Min. time w/GPU	1,243.5 ms	177.4 ms	4,865.0 ms	10,802.9 ms
Max. time w/GPU	1,398.7 ms	245.9 ms	16,053.1 ms	70,709.3 ms
Ave. time w/GPU	1,340.3 ms	206.6 ms	12,231.8 ms	42,948.1 ms
Min. Speedup	23.0	38.5	12.0	13.3
Max. Speedup	25.7	53.9	27.0	27.1
Ave. Speedup	24.0	45.9	15.1	16.4

Table 4 Performance of *alu4* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	16,219.7 ms	6,019.2 ms	60,460.7 ms	82,970.3 ms
Max. time w/CPU	20,661.1 ms	9,091.9 ms	93,772.2 ms	123,792.7 ms
Ave. time w/CPU	19,772.6 ms	8,386.4 ms	84,708.5 ms	113,263.4 ms
Min. time w/GPU	1,039.0 ms	132.1 ms	3,554.2 ms	6,502.0 ms
Max. time w/GPU	1,052.8 ms	154.2 ms	6,799.4 ms	11,178.7 ms
Ave. time w/GPU	1,044.1 ms	139.3 ms	5,438.7 ms	8,500.5 ms
Min. Speedup	15.5	43.4	13.6	15.2
Max. Speedup	19.7	66.0	19.2	20.3
Ave. Speedup	18.9	60.2	16.0	17.3

Table 5 Performance of *apex2* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	25,970.9 ms	8,020.1 ms	120,902.6 ms	156,958.0 ms
Max. time w/CPU	28,001.2 ms	9,104.0 ms	146,805.2 ms	182,581.4 ms
Ave. time w/CPU	26,439.2 ms	8,570.1 ms	137,966.6 ms	173,902.9 ms
Min. time w/GPU	1,109.1 ms	111.0 ms	5,639.5 ms	13,036.7 ms
Max. time w/GPU	1,298.5 ms	158.0 ms	11,483.9 ms	22,720.0 ms
Ave. time w/GPU	1,164.5 ms	135.2 ms	8,901.9 ms	15,913.5 ms
Min. Speedup	20.5	55.9	12.5	14.0
Max. Speedup	23.9	78.6	21.6	22.9
Ave. Speedup	22.7	63.9	16.1	17.6

Table 6 Performance of *pdc* initialized with fanout heuristic and 20% area free

	Mutation	Crossover	Fitness and Constraints	Total
Min. time w/CPU	31,476.9 ms	8,633.9 ms	145,464.6 ms	187,116.4 ms
Max. time w/CPU	32,764.1 ms	9,961.1 ms	186,007.6 ms	228,721.8 ms
Ave. time w/CPU	32,211.5 ms	9,339.5 ms	172,327.6 ms	214,013.6 ms
Min. time w/GPU	1,277.9 ms	145.2 ms	8,104.6 ms	14,250.6 ms
Max. time w/GPU	1,325.3 ms	197.7 ms	15,348.0 ms	29,715.3 ms
Ave. time w/GPU	1,296.7 ms	173.9 ms	12,094.0 ms	22,916.1 ms
Min. Speedup	23.9	43.7	12.1	13.6
Max. Speedup	25.5	65.8	17.9	19.6
Ave. Speedup	24.8	54.2	14.7	16.1

5 Conclusions

This research presents a significant acceleration of the partial fault tolerance method area-constrained partial fault tolerance using genetic evolution by employing NVIDIA CUDA to execute the algorithm on massively parallel graphical processing units. This speed up allows this method to be much more efficiently applied to larger circuits, and it will benefit from additional acceleration as the processing power of graphics processors tracks that of reconfigurable logic devices.

6 References

[1] J. Lach, *et al.*, "Efficiently Supporting Fault Tolerance in FPGAs," presented at the ACM International Symposium on FPGAs, 1998.

[2] F. L. Kastensmidt, *et al.*, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Design, Automation and Test in Europe*, 2005, pp. 1290-1295.

[3] D. L. Foster and D. M. Hanna, "Maximizing Area-Constrained Partial Fault Tolerance in Reconfigurable Logic," presented at the Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, California, USA, 2010.

[4] D. L. Foster, "Using Genetic Algorithms for Subset Selection for Partial Fault Tolerance in Reconfigurable Logic," in *The 2011 International Conference on Genetic and Evolutionary Methods*, Las Vegas, NV, 2011.

[5] B. Pratt, *et al.*, "Improving FPGA Design Robustness with Partial TMR," presented at the 12th NASA Symposium on VLSI Design, Coeur d'Alene, Idaho, 2005.

[6] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Test Conference, 2003. Proceedings. ITC 2003. International*, 2003, pp. 893-901.

[7] P. K. Samudrala, *et al.*, "Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *Nuclear Science, IEEE Transactions on*, vol. 51, pp. 2957-2969, 2004.

[8] O. Ruano, *et al.*, "A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs," *IEEE Transactions on Nuclear Science*, vol. 56, pp. 2091-2102, August 2009 2009.

[9] H. Quinn, *et al.*, "Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 2037-2043, 2007.

[10] F. Lima, *et al.*, "Designing fault tolerant systems into SRAM-based FPGAs," presented at the Design Automation Conference, 2003.

[11] NVIDIA Corp. (2011, Mar. 8). *NVIDIA CUDA C Programming Guide Version 4.1*. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[12] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: a Hands-On Approach*: Morgan Kaufmann Publishers, 2010.

[13] NVIDIA. (2010, Nov. 23). *Tesla C2050/C2070 GPU Computing Processor Overview*.

[14] A. Munawar, *et al.*, "Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 391-415, 2009.

[15] P. Krömer, *et al.*, "Many-threaded implementation of differential evolution for the CUDA platform," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[16] J. M. Cecilia, *et al.*, "Parallelization Strategies for Ant Colony Optimisation on GPUs," in *14th International Workshop on Nature Inspired Distributed Computing*, Anchorage, AK, USA, 2011.

[17] S. Tsutsui and N. Fujimoto, "ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[18] H. Bai, *et al.*, "MAX-MIN Ant System on GPU with CUDA," presented at the Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control, 2009.

[19] L. Mussi, *et al.*, "GPU-based asynchronous particle swarm optimization," presented at the Proceedings of the 13th annual conference on Genetic and evolutionary computation, Dublin, Ireland, 2011.

[20] H. Zhu, *et al.*, "Paralleling Euclidean Particle Swarm Optimization in CUDA," presented at the Proceedings of the 2011 4th International Conference on Intelligent Networks and Intelligent Systems, 2011.

[21] B. Rymut and B. Kwolek, "GPU-supported object tracking using adaptive appearance models and particle swarm optimization," presented at the Proceedings of the 2010 international conference on Computer vision and graphics: Part II, Warsaw, Poland, 2010.