Everything on the Chip: A Hardware-Based Self-Contained Spatially-Structured Genetic Algorithm for Signal Processing

Simon Perkins, Reid Porter, and Neal Harvey

Los Alamos National Laboratories, Los Alamos, NM 87545, USA,
{s.perkins,rporter,harve}@lanl.gov,
WWW home pages: http://nis-www.lanl.gov/{s̃imes,r̃porter,h̃arve}

Abstract. Evolutionary algorithms are useful optimization tools but are very time consuming to run. We present a self-contained FPGA-based implementation of a spatially-structured evolutionary algorithm that provides significant speedup over conventional serial processing in three ways: (a) efficient hardware-pipelined fitness evaluation of individuals, (b) evaluation of an entire population of individuals in parallel, and (c) elimination of slow off-chip communication. We demonstrate using the system to solve a non-trivial signal reconstruction problem using a non-linear digital filter on a Xilinx Virtex FPGA, and find a speedup factor of over 1000 compared to a C implementation of the same system. The general principles behind the system are very scalable, and as FPGAs become even larger in the future, similar systems will provide extremely large speedups over serial processing.

1 Introduction

Evolutionary algorithms (EAs) are perhaps the most general purpose practical optimization technique in use today. The same basic processes of evaluation, selection and recombination can be applied to any problem for which a fitness function and representation can be defined. The price we pay for this generality is that EAs typically take a very long time to find a solution for hard problems, as compared to more problem-specific techniques that make use of information about the nature of the problem being tackled.

The core process of almost all EAs involves performing a very large number of fitness evaluations. As a result, any way of speeding up fitness evaluations has significant consequences for how long the EA will take to solve a problem. Many researchers have looked at speeding up evaluation using reconfigurable hardware such as FPGAs (see [Higuchi et al., 1996] and [Sipper et al., 1998] for examples). Hardware fitness evaluation can provide a large speedup over software fitness evaluation in those cases where the fitness evaluation can be decomposed into many simple steps that can be carried out in parallel on the chip. For example, image processing algorithms are very time consuming to evaluate on a serial processor, but can be much more efficiently executed using a hardware pipeline.

A complementary approach to speeding up fitness evaluation is to use the hardware to carry out many fitness evaluations in parallel. In the limit, this would involve putting an entire EA population onto an FPGA. However, putting many evaluation units on a chip pushes very hard against the space limits of current FPGAs and so there are very few examples of this technique in the literature. One active area where we do see whole populations on a chip is the 'cellular programming' paradigm [Sipper, 1997]. CP solves the problem of fitting many units on a chip by making those units very simple — the basic individual in a CP system is a single cell of a cellular automaton, which implements only a very simple combinational logic function of its neighbors' states. The idea is that, although each component is very simple, complex large-scale behaviour can be achieved through local interactions between individuals in the evolving population. While this approach is very promising, it seems clear that it will not work for all problems. For more general EAs, we would like each individual to be capable of an arbitrary amount of computation by itself. Unfortunately, more complex individuals require more space per individual and space has always been at a premium on FPGAs.

However, the state of the art has recently taken a major leap forward with the arrival of the Xilinx Virtex FPGA [Xilinx, 1999]: a 'million-gate equivalent' part with a vast array of exciting hardware features that may well make it ideally suited to EA implementations. With the arrival of these chips it has now become feasible to place a reasonably sized population of reasonably complex individuals on a single FPGA part, with the following important consequence: if we choose our problem and our individuals appropriately we can then use the hardware to get a speedup in three significant ways:

- We get speedup by evaluating many individuals in parallel.
- We get speedup because with the extra space we can perform each of those evaluations in efficient hardware-parallel fashion.
- By putting the entire EA onto the chip, we get speedup by eliminating relatively slow chip-to-host communication and/or FPGA reconfiguration.

In this paper, we present preliminary results evolving a population of non-linear digital filters on a single Virtex chip to solve a non-trivial 1-D signal reconstruction problem. More important than the particular application though are the general architectural principles which can be expanded as FPGAs grow larger to provide greater and greater speedups compared to conventional processing.

2 Problem Definition and General Approach

2.1 1-D Signal Reconstruction

In the experiments reported in this paper, we look at the problem of reconstructing a pure signal from a digitized signal that has been corrupted with 'shot' noise: upsets that occur randomly at a constant expected rate and that

set the value of a sample to a totally random level. Such reconstruction problems occur in many practical situations, such as cleaning up signals sent over noisy channels in telecommunications systems. We would like our EA to design a filter that transforms the corrupted signal into a close approximation of the original one.

Stack Filters One important class of techniques for performing reconstruction makes use of 'stack filters'. A stack filter (SF) is a sliding window non-linear filter whose output at each window position is determined by applying a particular *positive* boolean function[1] (PBF) to a 'threshold decomposed' representation of the window.

The threshold decomposition process and subsequent filter operation is easily visualized as follows: take the 1-D string of values in the window to which the SF is being applied, and imagine them forming a 2-D 'wall' where the height of the wall at each window location corresponds to the value in that location. Then imagine taking 1-D horizontal slices of the wall, at every possible level (since we assume the signal values are discrete, there are only a finite number of such slices). Each slice gives us a 1-D string of boolean values where 'true' corresponds to 'inside the wall' and 'false' corresponds to 'outside the wall'. We apply the filter's PBF to each of these slices and find the total number of 'true' results we got from all the slices. This number is then the output of the filter at that window location. The fact that we use a positive boolean function, ensures that the output value is one of the input values.

Stack filters include as a subset such commonly used non-linear filters as the median filter, weighted median filter and other rank-order filters.

A number of researchers have used genetic algorithms to optimize stack filters, notably Chu in [Chu, 1990]. At least one researcher has also looked at evolving stack filters on an FPGA [Woolfries et al., 1998], but using the more 'conventional' technique of downloading and evaluating them one at a time on the chip.

Implementing Stack Filters in Hardware The threshold decomposition phase of stack filters can be time-consuming. For an 8-bit signal, there are 256 different levels at which to threshold, and hence each filter operation requires 256 evaluations of the PBF. Fortunately, there are more efficient ways of computing the same result. The technique we use was proposed by Chen [Chen, 1989] specifically for implementation in hardware. It relies on binary search and only requires a number of PBF evaluations equal to the number of bits used to represent the signal. The method assumes that the input values to the filter arrive as parallel bit-streams, MSB first. The PBF is first applied to the most significant bits of the input values, giving the MSB of the output value. If any of the input bits have a different value to the output bit, then that input stream is 'latched'

[1] A positive boolean function is a boolean function that can be written using just AND and OR and without negating any of the inputs.

at its current value. The PBF is then applied to the next bits from the streams, giving the next output bit, and the process repeats.

Stick Filters... Chen's method produces results equivalent to the proper stack filter algorithm, if the boolean function used is a positive boolean function. However, in the context of a genetic algorithm, it turns out to be fairly inconvenient to generate PBFs. The reason is that only a small fraction of all possible boolean functions for a given number of inputs are actually positive boolean functions, and it is time-consuming to check whether any particular boolean function is positive or not. In our experiments, we use a 5-element window, and so the boolean function can be defined by a truth table containing 32 1-bit values. There are $2^{32} \approx 4.3 \times 10^9$ possible such truth tables but in fact only 7581 of these represent positive boolean functions.

Our solution is simply to ignore the problem. We apply Chen's method using *arbitrary* boolean functions. The result is a stack filter if the boolean function happens to be one of the 7581 5-input PBFs, and 'something else' if it isn't. Exactly 'what' is difficult to describe concisely. The resulting filters are a rather strange class of non-linear digital filters that are a superset of stack filters. We call them 'stick filters', in reference to the way in which input bit stream values get 'stuck' if they disagree with the output value. The weird and wonderful space of stick filters is ideally suited to exploration by a genetic algorithm.

2.2 Genetic Algorithm Details

Representation Once the window size has been fixed, the operation of a stick filter is defined by its boolean function. For the 5-element windows we use in our experiments, an arbitrary boolean function can be represented by a 32 element truth table. It seems sensible to use a direct genomic representation here, and so the genome for an individual in our GA is simply a binary string giving the truth table for its boolean function, with the output value for an input of '00000' at the left end. Genetic operations are carried out directly on this representation.

Evolutionary Algorithm We use a fine-grained spatially-structured genetic algorithm that is similar to that used by Moshe Sipper in his cellular programming work [Sipper, 1997] and to other more conventional GA practitioners such as [Manderick and Spiessens, 1989]. A population of 48 evolutionary cells is distributed over an 6×8 grid on the FPGA. The cells are initialized with random truth tables. Each cell also maintains an error counter E, which is initialized to zero.

Once initialization is complete, evaluation begins. Each cell in the grid receives the same training data, consisting of a corrupted signal S^c and an uncorrupted version of the signal S^u, with the latter delayed by 2 time steps with respect to the former. At each time step (after the first 4), it applies the stick filter to a window consisting of the last 5 samples of S^c to generate a 'reconstructed' signal S^r. The absolute difference d between this output and the uncorrupted

signal's value at that time is computed and this value is added to E. After a fixed number of times steps τ, fitness evaluation is complete. This process is illustrated in Figure 1.

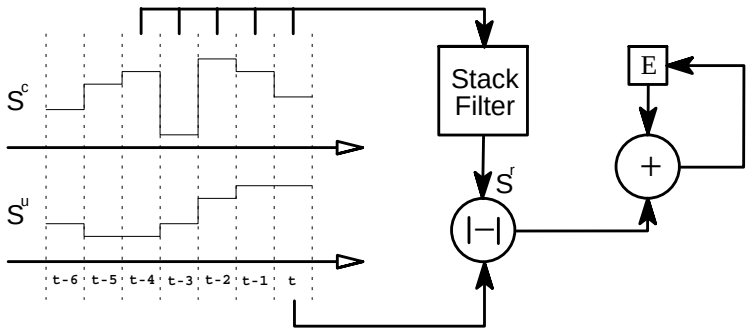

Fig. 1. Basic step of stack filter evaluation phase.

Once each cell's total error E has been determined, the GA goes into a breeding mode. Each cell compares its own error with that of each of its four neighbors to the north, east, south and west on the grid (toroidal wraparound is used at the edges). Once the fittest cell in its neighborhood has been found (this can be itself), uniform crossover is performed with the cell and its fittest neighbor. Note that if a cell is the fittest in its neighborhood then this uniform crossover leaves the cell unchanged. Point mutation is then applied to every element in the truth table of each cell with probability p_m. The effect of mutation is to flip that element's value.

Once breeding is over, the error counter is reset to zero, and the cycle repeats.

3 Simulation Experiments

3.1 Experimental Setup

A software simulator was written in Java, to test out the ideas behind the project. This functioned in an identical manner to that described above (except for running on a serial computer rather than an FPGA!). A signal consisting of a sine-wave with period equal to 20 time steps, quantized into 256 levels and corrupted by adding random shot noise was used as training input. Noise was added at an average rate (Poisson distributed) of 0.2 per time step. The effect of the noise was to drive the signal to a random value between 0 and 255, for that time step only.

The training period T was set to 2048 time steps, and the probability of a bit mutation p_m was set to 0.01 per bit.

3.2 Results

Figure 2 shows the way in which the error of the best individual in the population fell with generation number. The y-axis indicates the average error per sample of the best individual. Also shown is the error obtained on the same signal data, using a conventional 5-input median filter — a commonly used filter for removing impulse noise. The curves were obtained by averaging over five separate runs with different random number seeds.

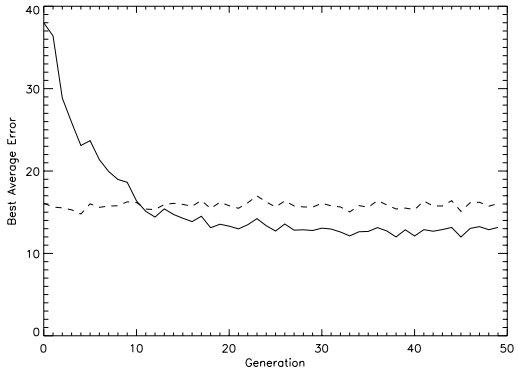

Fig. 2. Best average error vs. time graphs. The solid line shows the results from the best evolved filter, while the dashed line shows the results for a conventional 5-input median filter.

The graphs show clearly that the best of the initial random population fails to do as well the median filter, but after about 10 generations the best of the evolved filters begin to do consistently better than the median filter. There is some variation in the error scores from one generation to the next due to variations in the randomly generated input signals, but even so, the evolved filter always does better than the median.

A possible objection to this conclusion is that the 'best' evolved filter is chosen from 48 different candidates every generation, which could lead to an unfair bias in favor of the evolved filters. To counter this objection we took the most commonly evolved final genome (see below) and compared it head-to-head against the median filter on a fresh corrupted signal for an evaluation period of 30000 cycles. We also tested the filters on signals with different periods and different noise rates. Two-tailed paired sample t-tests were used to investigate (and confidently reject) the null hypothesis that the filters were performing identically on average. Table 1 summarizes these results.

An interesting fact is that in all the five runs performed for Figure 2, the same individual appeared most frequently in the final generation. This individual had the genome: 000000110001111100000111001111111.

Signal Period	Noise Rate	Median Filter	Evolved Filter	p
20	0.2	15.9	12.9	$\ll 0.001$
20	0.4	32.5	29.7	$\ll 0.001$
10	0.2	36.1	19.8	$\ll 0.001$
10	0.4	50.6	38.4	$\ll 0.001$

Table 1. Comparison of the best evolved filter with the median filter on various test problems. The third and fourth columns give the mean error per sample achieved during the tests. The final column gives the p-value for the null hypothesis that the mean performance of the filters is the same.

On closer inspection this turns out to be a positive boolean function (despite the fact that there was no explicit attempt to produce one) and seems to perform a 'weighted-center' median filter.

4 FPGA Implementation

Our design is targeted at the Annapolis Microsystems WildCard. This PCMCIA card contains a Xilinx Virtex 300 part and two independent banks of 256kB SRAM. The top level architecture for a cell is illustrated in Figure 3.

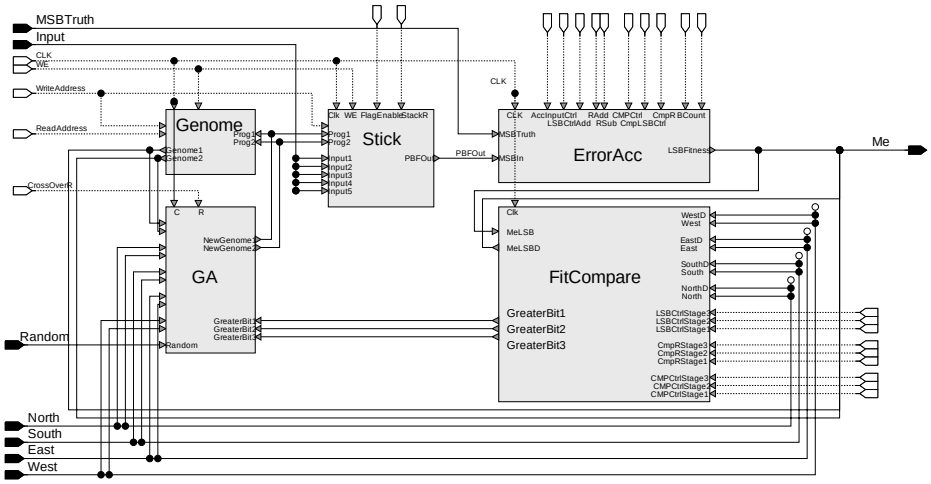

Fig. 3. Top level FPGA design.

At any one time a cell is in one of three modes. The mode is determined by control lines shown in Figure reffig:TopLevel with unshaded terminal nodes. Since all cells on the array are operating synchronously these control lines are

driven by a central controller common to all cells. A cell receives both genome and fitness information from its four neighbors illustrated by the shaded terminals 'North', 'South', 'East' and 'West'. The cell's own genome and fitness information are communicated to its four neighbors via the black terminal 'Me'.

In the first mode of operation, the stick filter's fitness is evaluated over 2048 input samples within the 'ErrorAcc' function block.

In the second mode of operation, this accumulated error is compared to error values of the cell's neighbors in the 'FitCompare' function block. The fittest neighbor is determined, and communicated to the 'GA' function block through the 'GreaterBit' control lines.

In the third mode of operation, the 'GA' function block makes use of a preloaded random bit stream entering through the 'Random' terminal to implement uniform crossover and mutation as described before, and to reprogram the stick filter.

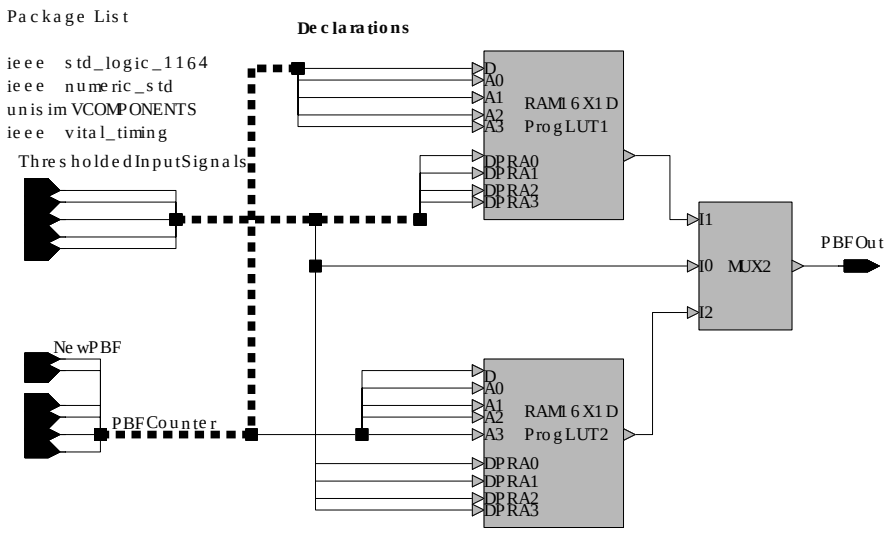

Fig. 4. Details of PBF implementation.

The core of a cell's signal processing power lies in the 'Stick' function block. The bit-serial nature of Chen's stack filter implementation results in an extremely compact architecture that is well suited to the Virtex FPGA. The complete filter can be implemented in just 3 Virtex CLBs (Configurable Logic Blocks). 1.5 of these CLBs are used to implement the cell's boolean function in the manner shown in Figure 4. The two programmable LUTs (configured as dual port RAMS) receive 4 of the 5 input samples through the 'ReadAddress' terminal and output to a multiplexer controlled by the 5th input sample. This configuration can implement any function of five variables. A very similar configuration is

found in the 'Genome' function block which holds a duplicate copy of the stick filter's 5-input boolean logic function.

Training data is pre-loaded into onboard memory in bit serial order, most significant bit first. On reset the cell array remains in an 'idle' configuration. At this time each cell's genome is configurable by the host processor. The array is activated when the host processor writes the number of desired generations to an on-chip register. The array then iterates for this number of generations, and, when complete, initiates an interrupt and returns to the idle configuration. At this stage the host program may once again read and write cell genomes.

A VHDL model of the GA has been both simulated and synthesized successfully for the Virtex 300 FPGA used on the WildCard. Table 2 analyzes the area requirements for the various components. Each CLB slice contains two 4-input look-up tables and two 1 bit registers. The controller moves each cell through its three modes of operation and also provides synchronization mechanisms for communicating with the host. The WildCard interfaces enable communication between the FPGA and onboard memory as well as to the host through the 32 bit CardBus.

Component	Number of CLB Slices	Percentage of Total
48 Cell Array	1904	62
Controller	43	1.5
WildCard Interfaces	549	18
Total	2496	81.5

Table 2. Space utilization on the Virtex 300.

Timing Experiments

For 8 bits/sample data, cells require 16 clock cycles to process one sample. Due to high fan out in synchronizing the cell array, clock rates were restricted to 20MHz, however higher rates are believed to be obtainable with more detailed design work. At this clock speed, the FPGA can run through one of the above experiments, using 48 individuals, run for 50 generations, and evaluated for 2048 time steps each generation, in an impressive 1.65 ms! In contrast, the Java simulator takes well over 5 minutes to achieve the same result.

Of course, the Java simulator was optimized for visualization rather than speed, so the core of the GA was re-implemented using C. The optimized C code took 1.8 s to perform the above experiment — still over 1000 times slower than the FPGA.

5 Further Work and Conclusions

At the time of writing the synthesized design has not actually been placed onto a real FPGA due to problems with our hardware, but we hope to remedy this in the near future.

It should be pointed out that the problem tackled here, while non-trivial, is not all that hard either — after all, a software version of the GA was able to find good solutions in just a few seconds! In order to demonstrate a real advantage we would like to extend the work here to look at optimizing more complicated types of signal and image processing functions that are intractably difficult to optimize in software. As an example, merely increasing the window neighborhood of our stick filter from 5 to 7 elements, increases the number of potentially useful positive boolean functions from 7851 to over 3×10^{10}! We can also imagine using FPGAs in situations where relatively simple problems such as this one must be solved extremely quickly. For instance, suppose we want to install an adaptive signal reconstructor on a communications channel that must re-evolve to match changing noise conditions (using a known signal sent over the channel as truth data), every few minutes.

As FPGAs continue to grow in size it will become easier and easier to fit larger and larger populations of more complex individuals on single chips and so we confidently expect this area of research to be a fruitful one.

References

[Chen, 1989] Chen, K. (1989). Bit-serial realizations of a class of nonlinear filters based on positive boolean functions. *IEEE Transactions on Circuits and Systems*, 36(6).

[Chu, 1990] Chu, C. (1990). The application of an adaptive plan to the configuration of nonlinear image processing algorithms. In *SPIE Proceedings - Nonlinear Image Processing*, volume 1247, pages 248–257.

[Higuchi et al., 1996] Higuchi, T., Iwata, M., and Liu, W., editors (1996). *Evolvable Systems: From Biology to Hardware: Proc. ICES '96*, volume 1259 of *Lecture Notes in Computer Science*. Springer.

[Manderick and Spiessens, 1989] Manderick, B. and Spiessens, P. (1989). Fine-grained parallel genetic algorithms. In Schaffer, J., editor, *Proc. 3rd Int. Conf on Genetic Algorithms*. Morgan Kaufmann.

[Sipper, 1997] Sipper, M. (1997). *Evolution of Parallel Cellular Machines*, volume 1194 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Sipper et al., 1998] Sipper, M., Mange, D., and Pérez-Uribe, A., editors (1998). *Evolvable Systems: From Biology to Hardware: Proc. ICES '98*, volume 1478 of *Lecture Notes in Computer Science*. Springer.

[Woolfries et al., 1998] Woolfries, N., Lysaght, P., Marshall, P., McGregor, S., and Robinson, G. (1998). Fast adaptive image processing in fpgas using stack filters. In Hartenstein, R. W. and Keevallik, A., editors, *Field Programmable Logic and Applications: From FPGAs to Computing Paradigm: 8th Int. Workshop*.

[Xilinx, 1999] Xilinx, I. (1999). Virtex 2.5v field programmable gate arrays. Advance Product Specification. Version 1.3.