Daniel Southern, Olivier Huber

Implementation of a Genetic Algorithm on an FPGA to Discover Efficient Sorting Networks

May 13th 2010

1 Background

Sorting sets of numbers is usually considered in the context of a software algorithm. However, when sorting functionality needs to be implemented in hardware, adapting a known sorting algorithm into hardware (i.e. quicksort, mergesort, etc.) may not be the most efficient or straightforward method. This is especially true when sorting relatively small sets of numbers, for example sets with 32 or fewer items. The alternative to an iterative or recursive algorithmic approach is to develop a static sorting network for the specific size of the set of numbers to be sorted.

1.1 Static Sorting Networks

A static sorting network can be represented as a list of *comparisons*, which specify two positions in the set to compare. Using a functional form, we can define a comparison function as $C(i,j)$. The behavior of the function is to swap the i^{th} and j^{th} items if the value of item j is less than the value of item i. By applying a list of comparison functions in a specific order, sorting networks can be formed which will result in a correctly sorted set of items given an arbitrary input configuration.

Optimal sorting networks are well known for very small problem sizes with fewer than 16 inputs. However, for 16, 32, and 64 input networks, it is not known whether more efficient networks can be developed. As recently as 1995, the best-known size of a 13-input sorting network was reduced from 46 comparisons to 45 comparisons using a software implementation of a genetic algorithm [1]. The possibility exists to develop novel, more efficient, sorting networks for larger input sizes.

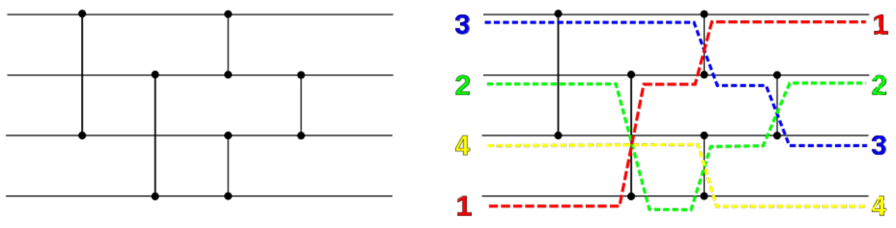


Figure 1: Visualization method for sorting networks. Image courtesy [2]

Figure 1 represents a visualization method for sorting networks. Each comparison is represented with a vertical line between the two positions being compared. Comparisons are applied in order from left to right, and positions are numbered with the first position at the top of the figure. It is possible for sets of comparisons to be performed simultaneously. This gives rise to two metrics for determining the performance of static sorting networks. First, the number of comparisons should be minimized, as each comparator represents extra hardware. Second, the number of parallel steps can determine the latency for receiving output from a sorting network in pipelined applications; the number of parallel steps should be also be minimized.

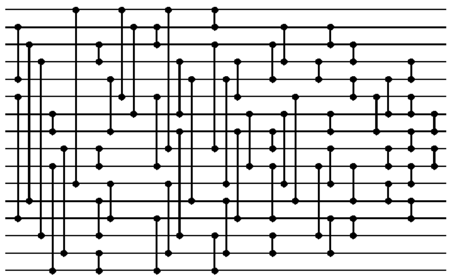


Figure 2: Visualization for a 16-input sorting network with 60 comparisons in 10 parallel steps. Image courtesy [1]

Figure 2 represents one optimized solution for a 16-input sorting network.

1.2 Genetic Algorithms

Genetic algorithms are employed to stochastically search through large solution spaces using the idea of *fitness* as a heuristic to guide the search. One way to frame a genetic algorithm is as follows:

In each iteration of the algorithm, a set of imperfect solutions, also called the population of solutions, is evaluated to determine how well each solution solves the problem at hand. Each member of the population is assigned a fitness via this process. After each member of the population is evaluated, individuals are chosen from the population stochastically, with the probability of being chosen tied to their relative fitness values such that more fit members have a higher probability of propagating from generation to generation. As members are copied from one generation to the next, random mutations are also added to some of the members in order to further explore the solution space. Mutations may be minor, which represents local search in the state space, or they can be radical to search the state space more broadly. The frequency and type of mutations can be modulated based on the overall progress of the algorithm.

1.3 Genetic Algorithms Implementations on FPGAs

By phrasing a search for efficient sorting networks in a genetic algorithm context, the problem appears in a form which lends itself to parallel optimization. Genetic algorithms are already intrinsically parallelizable due to their structure as a population of similar items which are each processed independently using the same functions. By implementing these functions in hardware multiple times, a speedup over software implementations can be immediately realized. Furthermore, the functions themselves may benefit from further parallelization.

Fortunately, a genetic algorithm designed to evolve efficient sorting networks benefits from both of these properties. Calculating the fitness of a particular sorting network involves checking its performance against many possible inputs, which immediately lends itself to parallelization.

1.4 Project Goals

By designing computational structures specifically for the sorting networks, it should be possible to realize huge speed increases versus software implementations which are delayed by non-specialized hardware, cache misses, etc. These overall speed increases may make it feasible to search larger problem spaces more effectively than was previously possible. The goal of this project will be to first reproduce optimal results for small problem sizes, and to ultimately tackle larger and larger problems in an attempt to discover novel sorting networks with more than 16 inputs.

2 Design Overview

High-level view

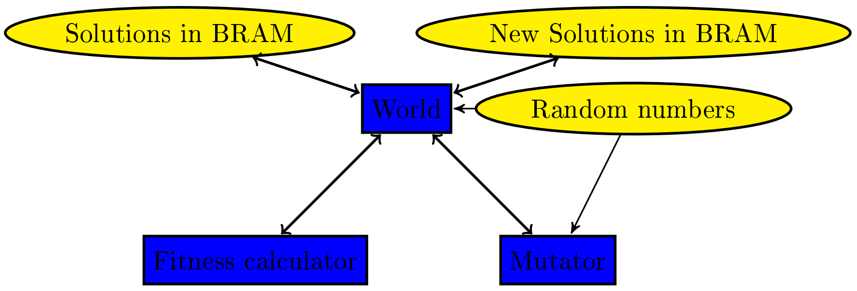


Figure 3: High-level design

The ellipse represents data and the rectangles represent modules.

In this architecture we have two important modules which are the **Fitness Calculator** module and the **Mutator** module. The first one takes as input sorting networks and random numbers to sort. It outputs a fitness value for the network.

The second one takes previous networks and produces new networks.

The central part is the **World** module that does all the necessary logical glue between each of the components. It should not be a time or resources consuming module.

We use LFSR (linear feedback shift register) as random number generator.

We spend most of our time and efforts in the **Fitness Calculator** and **Mutator** modules in order to make them as fast as possible and as small as possible.

2.1 Data Structures

In this design there are 2 primary data types, a data type to store the fitness of a network and the network itself.

2.1.1 Fitness

In order to maximize congruency with our C++ reference implementation, we chose to represent the fitness values as decimal numbers in the range [0.0, 1.0]. In hardware, we chose a FixedPoint representation with 2 bits for the integer component and 30-bits for the float component. This design decision ensures that range of values we want to capture are inside the range of values which can be represented, and choosing a standard 32-bit data width minimized confusion when transferring the fitness values via SceMi. The fitness type has the following Bluespec definition:

```
typedef FixedPoint#(2, 30) FitnessType;
```

2.1.2 Network

Choosing a representation for the network data type proved more challenging. We quickly realized that the number of bits in the network representation would preclude passing around monolithic network representations. Therefore, a *network chunk* data type was defined which held a small number of comparisons (i.e. between 1 and 5 comparisons). The number of comparisons in each chunk was left parameterized in our Bluespec code as a variable which we would investigate to optimize performance. The network chunks have the following Bluespec definition:

```
typedef struct{
  Vector#(chunkSize, Maybe#(Comparison#(netSize))) comparisons;
  Bool lastChunk;
} NetworkChunk#(numeric type netSize, numeric type chunkSize) deriving (Bits, Eq);
```

Choosing the chunk size can dramatically influence the design as it affects the data path widths between modules as well as the number of memory accesses required to read or write a network. Ultimately we found it preferable to specify a small chunk size of 2 and optimize the rest of the hardware to minimize dead cycles, compile time, and combinatorial path lengths.

2.2 Fitness Calculator Module

In a high level overview, the **Fitness Calculator** accepts a candidate network as input and produces a FixedPoint decimal value between 0 and 1 as an output which represents the fitness of the network. The fitness is calculated by applying every possible input to the sorting network and then determining the proportion of inputs for which the network correctly computes the sorted output. The **Fitness Calculator** module contains a bank of **Solver** submodules which complete the work of checking whether a particular input is correctly sorted by a network.

$$\textbf{Fitness} = \frac{\text{Number of inputs correctly sorted}}{\text{Number of possible inputs}}$$

Fortunately, Knuth provides a lemma which simplifies the problem of applying every possible input to the sorting network, known as the 0-1 principle [3]. The principle states that a sorting network which can correctly sort every possible input of 0's and 1's will be able to sort any arbitrary input. This greatly reduces the input space that we must evaluate, giving exactly 2^n possible inputs for a sorting network of size n. Furthermore, these problems lend themselves perfectly to representation as the set of unsigned binary numbers in the range $[0, (2^n - 1)]$.

For small sorting networks, the number of inputs is relatively small, and this approach allows fast evaluation. However, for larger networks, even this input space grows quickly, and we planned to move towards sampling just a subset of the possible inputs, or even simultaneously evolving populations of sorting networks and network inputs. We were unable to progress to sorting networks large enough to necessitate these changes.

2.2.1 Fitness Calculator Operation Overview

The top-level state machine keeps track of the current state of the system. When the system enters the *calculate fitness* state, a rule in the main module generates the appropriate requests for the population of networks to be streamed out of the **BRAM Memory** and into the **Fitness Calculator** module. As a network enters the **Fitness Calculator**, it is sent to each **Solver** in the bank of **Solver** modules. The number of **Solver** modules is a design parameter, as is maximized to best utilize the resources available on the FPGA. Each **Solver** module is assigned a possible input for the sorting network, which it sorts according the network being streamed through it. If the number of **Solver** modules inside the **Fitness Calculator** module is smaller than the number of possible inputs, then each network must be streamed through the **Fitness Calculator** multiple times. After each pass of the network through the **Fitness calculator**, the solvers are queried to accumulate a count of the number of correctly sorted results a particular network generates.

2.2.2 Solver Operation Overview

The **Solver** module is initialized with a n-bit binary input problem. It is then ready to accept a network stream which it will apply to the input problem. While the **Solver** applies the sorting network to the input, it simultaneously computes the expected sorted output. After the network has been streamed through the module, a 1-bit binary output is produced which indicates whether the network's output matched the expected sorted output. The **Solver** module is instantiated many times, and therefore warrants optimization in terms of both performance maximization and FPGA area usage minimization.

For example, as the **Solver** module applies the comparisons in each network chunk the input problem, we have a choice for how many should be applied in each clock cycle. Of course we attempt to apply as many as possible with the constraint that the chain of comparisons should not be the critical path in the design. The **Solver** operates by repeatedly applying comparisons to the input problem until the network chunk has been completed. When all of the comparisons in the final chunk of a network have been applied, the network's output is compared against the correctly sorted output to generate either a binary true or false output.

In our initial design proposal, we anticipated that the number of comparators applied serially per cycle in the **Solver** module would be a key factor in the system's performance, as well as the number of **Solvers**. We performed a design exploration to investigate the optimal number of comparators per **Solver** and the optimal number of **Solvers** and discovered that implementing a network chunk size of 2, along with 2 comparators per **Solver** gave the best performance. This is probably due to the fact that a network chunk can be processed in a single cycle in all modules, and data path widths are minimized, which helps maximize the number of **Solver** modules which can be instantiated on the FPGA. Ultimately we were able to instantiate 256 solvers, which utilized 95% of the LUTs on the FPGA. This demonstrates that parameterizing our design allowed us to effectively utilize all of the resources on the FPGA to maximize the performance of our design.

2.2.3 Sorter

When choosing which members of the population to copy into the next generation, we'd like to have a sorted list of the networks, with the networks with high fitnesses at the top of the list. Then we could draw randomly from the list, with a bias towards the networks with higher fitnesses. For example, if we ask the sorter for the 0^{th} element, it should return the index in the population of the network with the highest fitness.

Therefore, a sorting module was designed which accepts a fitness tagged with a population index for every member of the population; the fitnesses are stored and sorted. This facilitates biasing the reproduction step towards networks with higher fitnesses. When the next generation of networks is being computed, the top n fitnesses are drawn from the sorter to be propagated directly into the next generation both unaltered as well as through the **Mutator**, making up $\frac{2n}{\text{PopulationSize}}$ of the next generation. The rest of the next generation is created by mutating elements selected randomly from the current generation, with a bias towards higher fitness networks.

Implementation The sorter is implemented as a realtime insertion sort over the fitness values tagged with population indices. Since the fitness values are produced only occasionally by the **Fitness Calculator**, we can take advantage of the cycles in between by sorting each input as it arrives. The sorter is designed to only accept a new input when the current set of data in the sorter is already sorted. This allows the use of a binary search to locate the insertion point for the new element, which takes $O(\log(S))$ time for a population size of S. Next, elements in the array must be shifted down to accommodate the new element, which takes $O(S)$ time, with an upper bound of S cycles. As long as the **Fitness Calculator** requires more than S cycles to evaluate the fitness of a network, the sorter will not be a bottleneck to the system's performance. While this is not necessarily true for smaller problem sizes and larger populations (i.e. problem size of 8, population size of 2000), it is definitely true for networks of size 16 with population sizes up to 5000, and poses less and less of a problem with larger network sizes.

We implemented a 2-port BRAM to store the sorted values. This allows us to efficiently shift elements down the array by reading and writing simultaneously at adjacent positions in the RAM. Once the fitnesses for all members of the population have been inserted into the Sorter, we can bias a random selection of members of the population by choosing indices in the sorter biased towards the highest ranked fitnesses.

2.2.4 Fitness Calculator Design

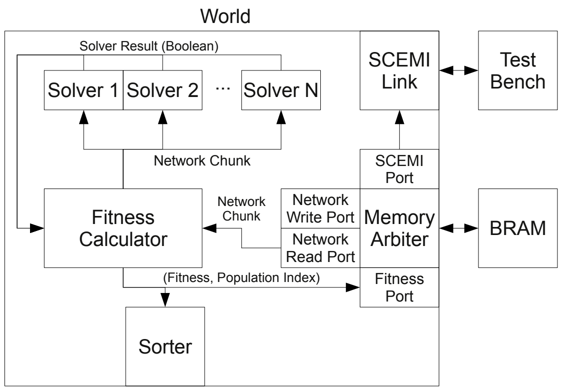


Figure 4: Fitness Calculator

2.2.5 Pseudo-Code of Fitness Calculator

The algorithms which define the operation of the **Fitness Calculator** are described below in psuedo-code.

Algorithm 1 Fitness Calculator

```
Require: N a pseudo sorting network
Ensure: Two comparisons are swapped
  f <- 0
  for $p = 0$ to $(2^{size(N)} - 1)$ do
    $p' = \text{ApplyNetwork}(N, p)$
    $p^* = \text{CalculateExpectedSolution}(p)$
    if $p' == p^*$ then
      $f$ <- $f + 1$
    end if
  end for
  return $\frac{f}{2^{size(N)}}$
```

2.3 Mutator Module

The **Mutator** module takes a sorting network as an input and produces another sorting network which may be slightly different from the original network through some transformation. This transformation is called a mutation, and the **Mutator** module will be capable of performing a variety of

Algorithm 2 ApplyNetwork

```
Require: $N$ a pseudo sorting network
Require: $p$ is a bit vector of length($N$)
Ensure: Bit vector $p$ is sorted according to $N$
  for $i = 0$ to length($N$) do
    $bit_1 = p\&(N.comp[i].c1)$
    $bit_2 = p\&(N.comp[i].c2)$
    if $N.comp[i].c1 > N.comp[i].c2$ and $bit_2 == 1$ and $bit_1 == 0$ then
      $p$ <- $p - (1 << N.comp[i].c2)$
      $p$ <- $p + (1 << N.comp[i].c1)$
    end if
    if $N.comp[i].c2 > N.comp[i].c1$ and $bit_1 == 1$ and $bit_2 == 0$ then
      $p$ <- $p - (1 << N.comp[i].c1)$
      $p$ <- $p + (1 << N.comp[i].c2)$
    end if
  end for
  return $p$
```

Algorithm 3 CalculateExpectedSolution

```
Require: $p$ is a bit vector of length($N$)
Ensure: The correct expected sorted output is calculated
  ones <- size($N$) - 1
  $p^*$ <- 0
  for $i = 0$ to size($N$) do
    if $p\&(1 << i)$ then
      ones <- ones - 1
      $p^*$ <- $p^*$ + (1 « ones)
    end if
  end for
  return $p^*$
```

mutations. In the context of sorting networks, a reasonable mutation may be to change the order of two comparisons, or to change one or both of the positions in a comparisons. Of course there are many more possible comparisons, and they can be composed to create even more complex mutations.

One interesting aspect of the **Mutator** module is that, as the mutations are supposed to random, it will require access to a pseudo-random source of numbers. This is represented in the design as a module from which we can simply request the next random number. For debugging purposes (in order to make our program's execution match up with the corresponding C++ program), this may be implemented as simply a list of pre-generated random numbers that we choose in order to emulate drawing random numbers in a deterministic, reproducible way.

The **Mutator** module provides a function interface which takes a Sorting Network. The module will then being processing the sorting network. A few cycles later, a new mutated Sorting Network will be available through another function at the output. The Control Logic references the Random Number Generator in order to select a Mutation to use. The Control Logic also takes a random number to pass to the **Mutator** which the **Mutator** will then use to perform the mutation.

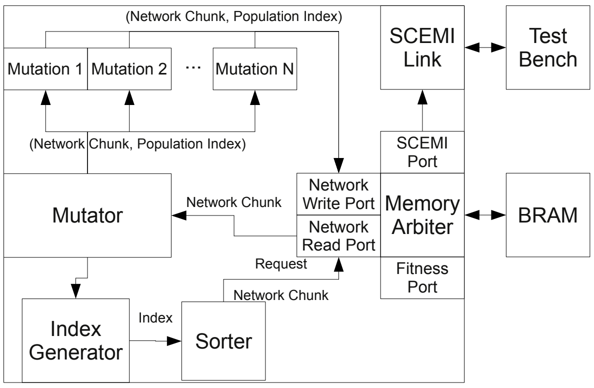


Figure 5: Mutator

2.3.1 Pseudo-code of mutations

Algorithm 4 Mutation 1

```
Require: N a pseudo sorting network
Ensure: Two comparisons are swapped
  RN1 <- RNG()
  RN2 <- RNG()
  N.comp[RN1] <- N.comp[RN2]
  N.comp[RN2] <- N.comp[RN1]
  return N
```

Algorithm 5 Mutation 2

```
Require: N a pseudo sorting network
Ensure: An index if one comparison is changed
  RN1 <- RNG()
  RNBIN <- Binary_RNG()
  if RBIN == 0 then
    N.comp[RN1].c1 <- RN1
  else
    N.comp[RN1].c2 <- RN1
  end if
  return N
```

Algorithm 6 Mutation 3

```
Require: N a pseudo sorting network
Ensure: A comparison is randomly changed
  RN1 <- RNG()
  RN2 <- RNG()
  RN3 <- RNG()
  N.comp[RN1].c1 <- N.comp[RN2]
  N.comp[RN1].c2 <- N.comp[RN3]
  return N
```

Algorithm 7 Mutation 3

```
Require: N a pseudo sorting network
Ensure: A comparison is randomly changed
  RN1 <- RNG()
  RN2 <- RNG()
  RN3 <- RNG()
  N.comp[RN1].c1 <- N.comp[RN2]
  N.comp[RN1].c2 <- N.comp[RN3]
  return N
```

Algorithm 8 Mutation 5

```
Require: N a pseudo sorting network
Ensure: Two comparison are swapped
  RN1 <- RNG()
  RN2 <- RNG()
  N.comp[RN1] <- N.comp[RN2]
  N.comp[RN2] <- N.comp[RN1]
  return N
```

2.3.2 Reproduction

Our initial designs chose networks uniformly randomly from the population to be propagated into the next population. This was accomplished via a LFSR which produces values in the range of the size of the population.

In order to bias network selection during reproduction, we created a module which would generate indices to be inserted into the sorter module described above. Based on the design of the sorter, the problem is reduced to generating indices in the range of the size of the population biased towards zero. We chose a polynomial probability density function $p(x, S)$ for the probability of selecting the x^{th} best element in a population of size $S + 1$, given by:

$$p(x, S) = \frac{3}{S^3}(S - x)^2$$

The coefficient $\frac{3}{S^3}$ is a scaling factor to make the integral $\int_{x=0}^{S} p(x, S) = 1$. The distribution is shown below for $S = 500$.

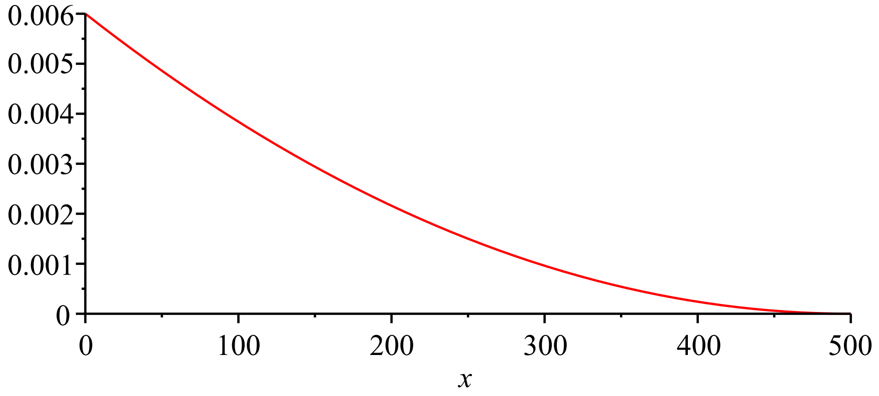


Figure 6: Probability of propagating the x^{th} best element into the next generation for a population size of 500

2.4 Memory Arbiter

A memory arbiter module was created to handle memory requests from three different sources. The memory arbiter handles marshalling and unmarshalling of the Fitness and Network Chunk data types. However, in our design we have kept this process simple by making sure each of these data types is less than 64-bits and can be read or written in a single memory access. This is not an issue since the rest of our design must be aware of the Network Chunk data-type as well, and we don't want our data paths to grow beyond 64-bits for performance reasons anyway.

The arbiter design is relatively simple. Read requests and write requests are enqueued in FIFOs as they arrive on the three different read/write ports. This means there are 6 total FIFOs which may contain access requests for the memory. We handle the read and write requests separately, keeping of track of whether the last memory access was a read or write and giving priority to the type of access which has happened least recently. This is also the strategy we use to mediate access between the three ports. The port which accessed the memory least recently is given highest priority access, and the port which accessed the memory most recently is given the lowest priority access.

Although this design may not be the most efficient, as it could be beneficial to prioritize certain types of memory access, it is a simple way to ensure that the system does not deadlock. We plan to aggressively optimize this component of the system, as dead cycles while we're waiting for memory access has already been identified as one of the largest performance problems in the current system design.

2.5 SceMiLayer

We are using SceMi to interact with the FPGA. We use it to transmit and receive information from the board.

We receive these informations from the board:

- The current best fitness
- The current number of cycles, iterations, number of cycles spend in each state
- We check if we have found a sorting network.
- If this is the case, we are getting this network, in order to visualise it.

We use SceMi to give some informations to the board:

- We are uploading the seeds for the LFSR.
- We are uploading the values of the CDF for the IndexGenerator module.

3 Debugging Strategy

We have 3 modules to test : the **Fitness Calculator** module, the **Mutator** module and the **World** module.

We used at the beginning a C++ program to search networks.

We managed to get a first working version pretty fast. In order to debug our design, we have use many `$display` command to be able to analyse what was going on during the simulation.

After we managed to get the first design working, we implement new features in bluespec, then check in simulation if everything is working fine immediately after we synthesize it on the FPGA and run tests on small networks to check if every is fine. Then we begin the cycle again.

4 Design Explorations

4.1 DDR Memory

In our preliminary design, the data structure we devised to represent a sorting network was one monolithic unit containing a vector of comparisons as well as a FixedPoint number representing the network's fitness. We were also storing the population of networks in a vector of registers. For extremely small networks (i.e. with 4, 5, or 6 inputs and 5, 9, 12 comparisons respectively), this data structure remained a manageable size, i.e. less than 128 bits. As we progressed to larger networks, two problems were introduced.

- **Storage** The number of registers required to store a sufficiently large population of networks grew too quickly. This led to both extremely long compile times, and usage of a large area of the FPGA.
- **Data Path Width** The width of the data paths was also a major concern. Sending the entire network at once between the modules in our design increases the complexity of the routing the compiler must perform.

In order to address these problems, we undertook major architectural changes to make our design less sensitive to the size of the sorting network. This involved splitting up the network data structure into smaller chunks. Rather than storing all of the comparisons in a single data structure, we introduced a parameterized data structure which holds a small number of comparisons (i.e. 1 - 5). We also decided to store the fitness value separately, as the data paths for the fitness values are separate from the comparisons in many parts of the design. The biggest change, however, was to move storage of all information from registers on the FPGA into the DDR2 memory of the FPGA.

This required the introduction of a memory arbiter module which presents separate memory access interfaces to 3 separate types of interactions with the memory.

- **SCEMI** A port was created to give SCEMI read-only access to the memory. This allows our TestBench C++ program to query the fitness values of the population to calculate statistics such as maximum fitness, average fitness, etc.

- **Network Chunk Access** The **Fitness Calculator** and **Mutator** modules never operate simultaneously, so a single network chunk interface sufficed. This interface allows both reading and writing of network chunks.

- **Fitness Write Access** The **Fitness Calculator** module writes the fitness of each network into the DDR2 memory for SCEMI to read.

As the **Fitness Calculator** module computes the fitness of a network, the network is streamed into the module in chunks. The chunks are requested ahead of time to maximize throughput in the face of the memory delay. Furthermore, each network must be streamed from memory into the **Fitness Calculator** several times, as we are not able to apply the network to every possible input in a single pass.

The **Mutator** module only needs each network to be streamed in once. However, the networks are not necessarily read in order of the memory, as during reproduction one network may be included in the next generation more than once, while another network may not be included at all. The **Mutator** module receives all of the chunks from a network in order, and eventually emits all of the chunks of the new (mutated) network, but not necessarily in order, as the amount of state the **Mutator** module needs to maintain was minimized, but the **Mutator** may need to perform an action such as swapping comparisons in the first and last chunks of a network. The intermediate chunks will be emitted form the **Mutator** unaltered, but the first and last chunks cannot be emitted until the entire network has been streamed in.

When we moved to the DDR2 storage, we faced a major regression in performance. Since we were requesting only one chunk at a time from the DDR2, and we were only able to write one chunk every 4 cycles. After digging into the module and interface with the DDR2, it appears that we could try to cache the data that we requested in order to circumvent the latency in come cases, but ultimately DDR2 would still be the bottleneck in the performance of our design without more serious architectural changes. Furthermore, we wanted to concentrate our design efforts on the components which were more specific to the project rather than designing around the characteristics of the memory.

Through further consideration about the amount of storage space we would need to store networks as we moved to larger networks, we approximated that 1MB of storage would an upper limit throughout this work. The FPGA we're using has approximately 3MB of BRAM available. Furthermore, the BRAM memory provides single cycle delay for read and write, and provides an interface which is simpler than the DDR2 burst interface. The calculations to estimate memory usage are reproduced below:

Memory Requirements For a network of size n with c comparisons and a population size of S, we calculate the memory requirements:

- **Network Indices:** Must represent numbers [0, n - 1]. Size is TLog$\#(n)$.
- **Comparator:** A comparison consists of two Network Indices, which has size $2 \cdot$ TLog$\#(n)$
- **Network:** A network consists of c comparisons, which has size $2c\cdot$ TLog$\#(n)$
- **Population:** The population consists of S networks, and is instantiated twice to allow copying elements from one generation to the next. This has size $(2S) \cdot (2c) \cdot$TLog$\#(n)$.
- **Population Fitness:** Each fitness is size 32-bits, and we must store S fitnesses.

The total size in bits is then approximated by:

$$4Sc\text{TLog}\#(n) + 32S$$

For a network of size 16 and a population of size 10,000, we calculate needing approximately 1.2MB of storage.

4.1.1 Miscellaneous Optimizations

When our code was correct, we then tried to decrease the average time for evaluating the fitness of a network.

One important element are dead cycles, which can easily happened with FIFOs.

Since we have already some big combinatorial paths in your design, which prevent us from running it at 100 MHz, we tried to use BypassFIFOs to decrease the number of dead cycles.

5 FPGA usage

We compiled for the fpga for up to size 16.

Size	Comparisons	Population Size	Mean Iterations	Mean Time (s)
8	19	1000	893	8
9	25	1000	4170	24
10	29	1000	101209	596
11	35	1500	37127	566

Table 1: FPGA Results

Size	Comparisons	Slice Register Utilization (%)	Slice LUTs Utilization (%)
8	19	25	66
9	25	28	74
10	29	29	75
11	35	30	77
12	39	31	80
13	45	31	86
14	51	32	89
15	56	33	92
16	60	34	95

Table 2: FPGA Utilization for designs with 256 solvers

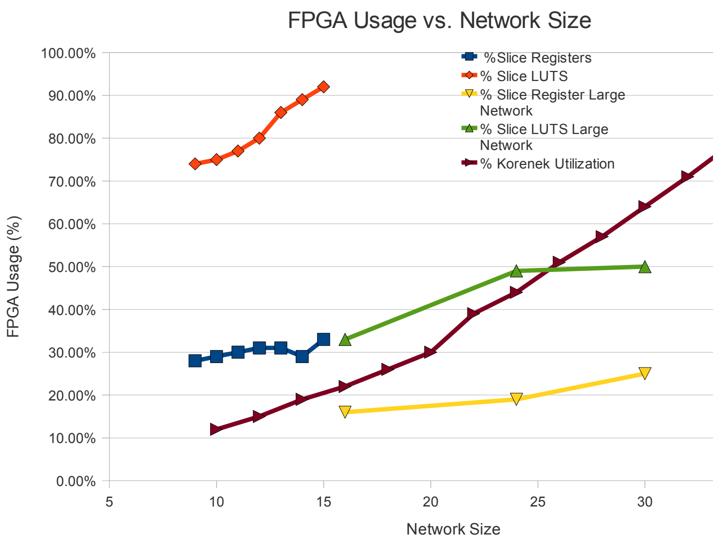


Figure 7: Comparison of FPGA Utilization vs. Network Size in Several Parameterizations in Bluespec and in the Work of Korenek [4]

Module	Slices	Slices Register Utilization	Slices LUTs
Bridge	6920/36353	1376/26488	7174/61928
NetworkFinder	698/27042	938/18950	1631/51860
FitnessCalculator	760/25073	43/16284	1099/47775
Solver	51/92	63/63	110/180
Comparator	41	0	70
Mutator	150/725	6/1120	257/1329
Mutation1	163	331	309
Mutation2	143	233	250
Mutation3	14	26	28
Mutation5	106	213	109
Sorter	259	215	555
Scemi	1359	2780	2078

Table 3: Area Utilisation for a network of size 16 with 256 solvers

If we compare our result against Koronek works and our C++, we have this results :

Network Size	Us	Korenek	C++ program
8	5 μs	5 μs	71 μs
12	15 μs	81 μs	4300 μs

Table 4: Evaluation time of one network

6 Conclusion

In this work we successfully created a parameterized Bluespec module to evolve efficient sorting networks. We have demonstrated that our design is relatively flexible and allowed us to perform some design exploration to maximize performance in the parameter space. Furthermore, the biggest advantage of this design is that we can leverage all of the resources on the FPGA to maximize the speed of the genetic algorithm. We have demonstrated that our design evaluates the fitness of individual networks up to 10 times more quickly than previous similar work on an FPGA.

Unfortunately, despite the ability of our design to evaluate networks more quickly than Korenek, we failed to produce sorting networks as large as Korenek was able to produce. We attribute this to the limitations of our implementation of the genetic algorithm. Specifically, we feel that a more sophisticated reproduction mechanism which maintains higher genetic diversity would be desirable, along with a better mechanism to modulate the rate of mutation according to the progress of the algorithm.

6.1 Lessons Learned

If we were to undertake a similar project in the future, we would choose to spend more time thinking about the problem abstractly rather than spending the majority of our time maximizing performance on the FPGA. Although we were able to achieve satisfactory performance, our algorithm ultimately failed to produce the interesting results we were hoping for. It appears to be the case that a more sophisticated algorithm can afford increases in search speed much larger than we were able to achieve by even doubling the speed on the FPGA.

For reference, the results from [4] are reproduced below.

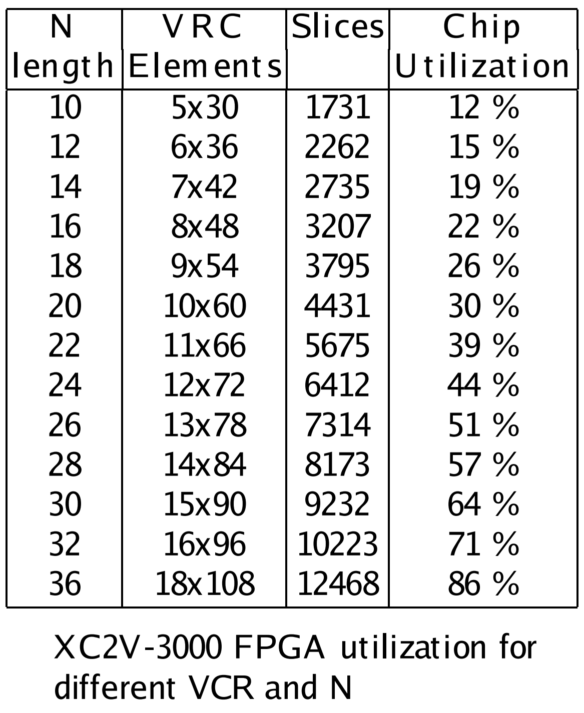
N length	VRC size (elements)	# Perfect solutions	Average num. of generations	Standard deviation	Evaluation time of one candidate
4	2x8	80	94	2.55556	512 ns
6	3x16	80	458	31.44444	1.28 us
8	4x16	80	2217	24.66667	5.12 us
10	5x32	80	6378	65.66667	20.48 us
12	6x32	76	8673	666.88889	81.92 us
14	7x32	75	11322	718.55556	327.67 us
16	8x32	66	19467	477.44444	1.31 ms
18	9x64	20	25306	3732.77778	5.24 ms
20	10x64	17	31344	150.00000	20.97 ms

Table 3. Sorting networks evolved in FPGA

N length	VRC size (elements)	Number of generation	Evaluation time of one candidate	Total time of evolution
22	11x64	4044	83.89 ms	5.6 min
24	12x64	4804	335.54 ms	26.9 min
26	13x64	10027	1.342 s	3.7 h
28	14x64	13483	5.368 s	20.1 h

Table 4. Large sorting networks evolved in FPGA

Figure 8: FPGA results



N length	VRC Elements	Slices	Chip Utilization
10	5x30	1731	12 %
12	6x36	2262	15 %
14	7x42	2735	19 %
16	8x48	3207	22 %
18	9x54	3795	26 %
20	10x60	4431	30 %
22	11x66	5675	39 %
24	12x72	6412	44 %
26	13x78	7314	51 %
28	14x84	8173	57 %
30	15x90	9232	64 %
32	16x96	10223	71 %
36	18x108	12468	86 %

XC2V-3000 FPGA utilization for different VCR and N

Figure 9: Chip utilisation

References

[1] H. Juillé, "Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces," in *Proceedings of the 6th International Conference on Genetic Algorithms*, pp. 351–358, Citeseer, 1995.

[2] O. Sigvardsson, "Simple sorting network full operation." Wikipedia. http://upload.wikimedia.org/wikipedia/en/thumb/9/9b/SimpleSortingNetworkFullOperation.svg/1000px-SimpleSortingNetworkFullOperation.svg.png.

[3] D. Knuth, *The Art of Computer Programming*, vol. 3 - Sorting and Searching. Addison-Wesley, 1973.

[4] J. Korenek and L. Sekanina, "Intrinsic evolution of sorting networks: a novel complete hardware implementation for FPGAs," in *Evolvable Systems: from Biology to Hardware: 6th International Conference*, (Sitges Spain), p. 46, Springer Verlag, 2005.