Synthesis of a Systolic Array Genetic Algorithm

G. M. Megson*and I. M. Bland†
PEDAL
Department of Computer Science
University of Reading
Whiteknights
Reading RG6 6AY
U.K.

Abstract

The paper presents the design of a hardware genetic algorithm which uses a pipeline of systolic arrays. Demostrated is the design methodology, where a simple genetic algorithm expressed in C source code is progressivly re-written into a recurrence form from which systolic structures can be deduced. The paper extends previous work by the authors by introducing a simplification to a previous systolic design.

1 Introduction

The Genetic Algorithm (GA) is now a well established technique for search and optimization [3]. Inherent to the technique is a rich source of parallelism, which exists at many levels, and this has attracted the attention of the parallel research community. By using parallelism it is possible to speed up the execution of the algorithm. Thus the technique can be applied to larger problems or deliver acceptable performance in real-time applications.

Attempts to this end have been made on both general-purpose parallel architectures and, more recently, on special purpose processing devices. The attraction of custom-build hardware lies in both the exploitation of massive concurrency and the speed-up obtained when the machine architecture is tailored to the algorithm.

We have been using systolic array techniques [4] to design a special-purpose GA accelerator [5]. Our design results from a systematic application of regular array synthesis techniques to a C code version of the GA. These techniques express the algorithm as a set of uniform recurrence equations, from which the data-dependencies can be extracted. By expressing these recurrences in a paticular way, systolic architectures can be derived which exploit parallelism using locally connected structures, suitable for silicon implementation. This paper is intended to demonstrate this design path.

The paper is organized as follows: Section Two introduces Genetic Algorithms and discusses related work. Section Three describes the design process with conclusions in Section Four.

* G.M. Megson@reading.ac.uk
† I.M.Bland@reading.ac.uk

2 Genetic Algorithms

2.1. Theory

Genetic algorithms are loosely based on the mechanics of natural selection. A population of candidate solutions (Chromosomes) is held and interact over a number of iterations (Generations) to produce better solutions. The solutions are encoded as strings of characters from an n-ary alphabet. For the canonical GA, n=2 and the chromosomes exist as binary strings. Driving the process is the fitness of the chromosomes, which relates the qualitative ‘goodness’ of a candidate solution in quantitative terms. The fitness function encapsulates the problem-specific knowledge. The fitness is used in a stochastic selection of pairs of chromosomes which are ‘reproduced’ to generate new solution strings. Reproduction involves Crossover, which generates new children by combing chromosomes in a process which swaps portions of each others genetic material. A second reproduction operator, Mutation is then applied to the new chromosomes. Mutation randomly changes genes and is used to introduce new information into the search.

The canonical GA uses the Roulette-Wheel technique to select chromosomes for reproduction. Roulette Wheel selection operates by first finding the total fitness of the population. This is used to normalized the population (so that

the total fitness=1) and a random number is generated between 0 and 1. Each chromosome is taken in turn and its fitness is added to a running sum. The chromosome (i) which satisfies the condition $\sum_{j=1}^{i} fitness_j > RND$ is passed for reproduction with a second chromosome selected in an identical manner. It is clear that the fitter the chromosome, the greater the chance of selection

Crossover is used to construct new 'children'. In the canonical GA, One-point crossover is used. An alternative scheme is Uniform crossover [7] which makes a random decision *for each gene* whether it will form part of the child.

Mutation acts to change a gene which, for a binary chromosome, simply involves flipping a bit from one to zero or visa versa. It is used sparingly to expediate the exploration of the search space.

2.2. Related Work

Previous designs for hardware genetic algorithms have been contributed A good overview of the field is given by Scott [6]. A general assumption adopted by these designs is that the problem-specific fitness function should be the candidate for hardware implementation as this usually accounts for the majority of processing time. We take a slightly different approach [5]. We believe that it is the genetic operators which are the ideal candidate for hardware implementation due to their regularity and generality. In general fitness functions are best performed by general-purpose processors at a higher level of abstraction, using a coarse grained parallelism to reduce the bottleneck associated with this task.

We employ seven systolic arrays to perform the genetic operators. These are arranged as a macro-pipeline with the chromosomes flowing uni-directionally through. The fitness of each chromosome is evaluated externally and precedes the genetic portion of the chromosome. The evaluation of the fitness before the chromosome enters the macro-pipeline removes the need for fitness evaluation hardware and enables this task to be divorced from the operators. Throughout the design process we have chosen to express a particular dimension, the lengths of the chromosomes, as time. This results in arrays which are scaled solely by the population size (N). Two advantages result. Firstly the architecture is capable of processing chromosomes of different lengths without re-implementation. For problems with very long chromosomes, the architecture also becomes very efficient in terms of area. Secondly, the population of a large global GA can be broken up and distributed as demes in a coarse-grained parallel GA. Each deme being processed on an individual GA processor with some interconnection scheme implementing Migration [2].

```
tf=0;
for (i=0;i<N;i++) {
  tf += chr[i].fit; }
```

(a)

```
tf[0]=0;
for(i=0;i<N;i++) {
  tf[i+1] = tf[i] + chr[i].fit; }
```

(b)

Figure 1. C code for Total Fitness

3 The Systolic Array Genetic Algorithm

The seven arrays are, in order, Roulette Wheel, Select, Sort, Match(1), Match(2), Crossover and Mutate. The first two arrays, Roulette Wheel and Select and the last two Crossover and Mutate are used by our revised design and will be discussed below. The three remain arrays, Sort and the two Match arrays, result from a previous attempt to overcome a fundamental difficulty associated with synthesizing arrays for the GA. As discussed above, Roulette Wheel selection makes a random but biased selection of chromosomes for reproduction. The random element ensures diversity within the search and opens up areas of the search space for exploration. Synthesis techniques are defeated by randomness because it is run-time dependent. The construction of the data-dependency graph is made using a *static* analysis of the algorithm. A second problem also associated with selection is the location of selected chromosomes. The algorithm dictates any chromosome can be paired with any other. In a systolic system these chromosomes need to be spatially adjacent.

These difficulties have been addressed by modifying the underlying algorithms so that synthesis techniques can be applied. In our previous work we introduced a Sort and two Match Arrays to solve these problems. A prior re-apprasial of the algorithm has removed these and this re-apprasial is a contribution of this paper. We shall proceed to present the synthesis process for the new architecture.

3.1. The Roulette Wheel

Synthesis starts with original algorithm. The algorithm is re-written as a set of recurrence relations and re-expressed so that a particular element of parallelism can be exploited. The process is iterativly adjusting the architectures which result. The analysis starts with the GA expressed in C. Fig 1a gives the first code segment of the GA we wish to transform. The code is a simple loop which calculates the total fitness of the population. This and the subsequent code

```
tf=0;
for (i=0;i<N;i++) {
  chr[i].fi=chr[i].fit/ tf; }
```

(a)

```
tf[0]=tf /*from the summation array*/
for (i=0;i<N;i++) {
  chr[i+1].fit=chr[i].fit/ tf[i]
  tf[i+1]=tf[i]; }
```

(b)

Figure 2. C code for Normalization

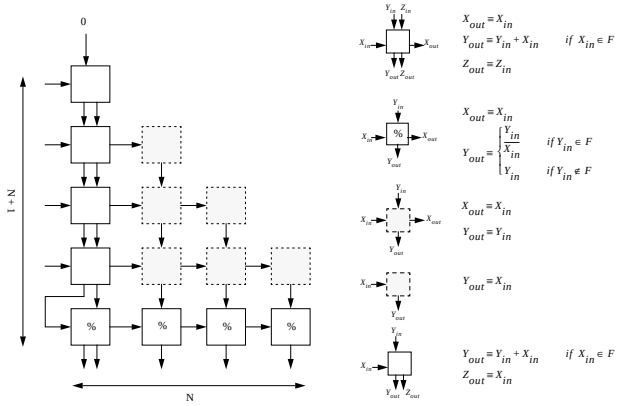

Figure 3. The Folded Roulette Wheel Array

section, normalization (Fig 2a), will form the basis of the Roulette-Wheel array. To extract the recurrences the variables need to be fully indexed, so that they can be represented geometrically as occupying a position in p dimensional space. A further dimension of time is used to reflect the ordering of computation. Thus p+1 dimensions are required and we term this co-ordinate system Space-Time [4]. Our aim is to project the Space-Time representation of the algorithm into a maximum of three dimensions (two spatial and one time), thus allowing planar array structures.

The code has one loop so p=2 and each variable requires one index (the second dimension being time). Whilst the variable `chr[i].fit` has the index i, `tf` is not indexed and therefore also not in single assignment form. Using i as the index is suggested by the for-loop. The accumulating total fitness is passed down the array using pipelining [4]. This is reflected in the code by incrementing the i index on the assignment. Although this will impose a strict sequential ordering on the computation, it allows a locally connected systolic structure. The code is re-written to give Fig 1b.

The second code segment is similar to the above (Fig 2a). This code uses the total fitness, calculated previously, to normalize the fitness values. The variable `chr[i].fit` is indexed by i and so already in single assignment form. The variable `tf`, the total fitness, requires a index term. `tf` remains constant throughout the loop and could have a constant index, such as zero. This will result in recurrence which is non-uniform. The effect being that a non-local broadcast of `tf` would be required in the final architecture. We can manipulate these (affine) recurrences using pipelining. Using the index i and pipelining in this direction results in a uniform recurrence. The new code is given in Fig 2b. The two sections now need to be joined together.

Each fitness value will be required twice by the hybrid architecture. A schedule for the computation is derived to ensure that the data dependencies are met and all the required data arrives at each site of computation at the correct time. The dependencies indicate the cell at row i is active at time i and the total fitness becomes available at time $i = N$, for normilizasion. This imposes a stagger on the incoming fitness values. The first fitness value is required at time $i = 1$ and again at time $i = N$, implying the use of $N \times (N - 1)$ delay cells (cells which perform no computation) between the two resulting architectures to ensure synchronization.

We have employed the principle of Space Folding [4] here by re- mapping two elements (in space) onto one location. The axis of the fold runs from top left to bottom right, including the first fitness summing column but excluding the second normalization column. Data is turned through 90 degrees and emerges from the bottom of the array. Fig 3 describes the new folded array which is an $N \times (N + 1)$ triangular array. The left-most column is the first summation array, as well as including delay cells re-mapped onto this location by the fold. The bottom row performs normalisation. Both use N cells, suggested by the N bounds on their loops. The connection between the two is local, length invariant of N and so systolic; a direct result of using the fold. The fold has also reduced the number of the cells in the design but at the cost of increased cell complexity.

3.2. The Select Array

The first array has normalized the fitness values in preparation for selection. Fig 4a is the C code for selection itself. In a software GA, selections would normally be made in pairs with reproduction proceeding on these pairs directly. Here it is best to perform all of the selections together and then progress to reproduce the population as a whole. The C code reflects this by nesting the selection operator within a loop of size N. Inside this loop a random number in the range of 0 to 1 (the ball value) is chosen by a Pseudo Random Number Generator (PRNG). Each normalized fitness is then summed in turn until the running sum exceeds the ball value. As the code stands it cannot

```
for (i=0; i<N; i++) {
  ball = rand() % tf;
  j=0;
  running_sum=0;
  while ((r_sum+chr[j].fit)<ball) {
    r_sum+=chr[j].fit;
    j++; }
  selected[i]=j; }
```

(a)

```
for (i=0; i<N; i++) {
  ball = rand() / INT_MAX;
  for(j=0; j<N; j++) {
    if (ball - chr[j].fit< 0) {
      selected[i]=j;
      ball=FLOAT_MAX;
      } else {
      ball=ball - chr[j].fit; }}}
```

(b)

```
for (i=0; i<N; i++) {
  ball[0][i]=rand() / INT_MAX;
  for(j=0; j<N; j++) {
    if (ball[j][i]-chr[j][i].fit< 0) {
      select[j][i+1]=chr[j][i].gene;
      ball[j+1][i]=FLOAT_MAX;
    } else {
      ball[j+1][i]=ball[j][i]
                   -chr[j][i].fit;
      select[j+1][i]=select[j][i]; }
    chr[j][i+1]=chr[j][i]; } }
```

(c)

Figure 4. C code for Selection.

be synthesized. The guard on the while is a run-time dependent variable and therefore cannot be used by a static analysis of the algorithm. The normalization of the fitness values performed in the previous stage implies the invariant $\sum_{i=1}^{N} fitness_i = 1$. With $ball \leq 1$, selection can be guaranteed with N iterations. The while loop is therefore replaced with a for loop bounded by N and the run-time dependency is removed. Although at the price of an algorithm which is less efficient. We have also used subtraction (from the ball value) rather than addition to test for selection. This is functionally identical but facilitates the construction of the hardware. Fig 4b gives the new code ready to be fully indexed and re-written in single assignment form.

The code has two nested loops, implying a Space-Time representation in three dimensions. Each variable requires 2 indices to be fully indexed. The loops are bounded by

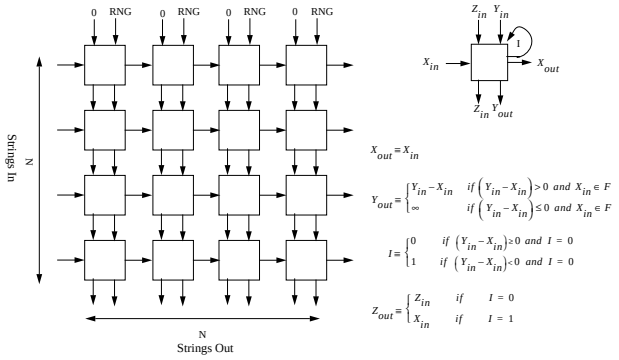

Figure 5. The Revised Select Array

the population size N so an $N \times N$ array is expected. The inner loop corresponds to one selection and one column of the final array. The ball value is pipelined down this column using j as the index, each cell modifies the ball value by subtracting the normalized fitness. The value `ball[0]= RND`, where $0 \leq RND < 1$, is the input to the column at the top.

In our new architecture we adopt a different approach for selection than before. A value (`select`) is pipelined down the column, in the same direction as `ball`. This approach guarantees `select` will visit every cell in the column and, by definition, the cell which selects the chromosome, even though before run-time it is not known which this will be. The invariant that only one selection can occur in each column is exploited by overwriting select with the actual chromosome (minus its fitness field, which is no longer required) and passing it through the remaining cells. The time projection of this dimension implies the chromosome is following behind (we omit this in the code) so we copy it into `select` a gene at a time and pipeline the result out.

The outer loop requires another index to fully index the variables and ensure single assignment. The two indices now represent axes in space. The first index j represents the vertical axis which runs down the columns with the horizontal axis, i, defining the N different selection columns. The order of indexing is reverse from traditional Cartesian coordinates, which can be confusing but is nevertheless consistent and results from the order of index assignment. As each column uses a mutually exclusive ball value, `ball` does not require pipelining in i and i is used only to ensure single assignment (to localise the cell in space). A copy of the chromosome (with fitness field) is also pipelined horizontally so multiple selections can be made. Fig 4c gives the modified single assignment code for selection with Fig 5 depicting the modified Select array.

```
for (i=0; i<N; i+=2) {
  for(j=0;j<NGENE; j++) {
    if (randval()) {
      chr[i].gene[j]=chr[i+1].gene[j];
      chr[i+1].gene[j]=chr[i].gene[j];
      } else {
      chr[i].gene[j]=chr[i].gene[j];
      chr[i+1].gene[j]=chr[i+1].gene[j];
      }}}}
```

Figure 6. C code for Crossover

3.3. Crossover

Crossover and the following Mutation arrays act only on the genetic portion of the chromosome. We rewirte the code into a suitable form (Fig 6) which applies Uniform Crossover to adjacent chromosomes. We deal with `randval()` as a special case. We cannot assume `randval()` is a constant number. The notion of a constant random number makes no sense here. Instead we use an arithmetical peuedo random number generator which exists, conveniently, as the recurrence $X_{n+1} \equiv \lambda_1.X_n + \mu_1(modP)$. In our scheme we can simply unroll the recurrence directly into the array cells. Seeding from above. Details of this method are given in detail elsewere [1]

We unroll the recurrences into a single column of N/2 cells which accepts pairs of chromosomes [5]. The array has been reduced from an $N \times L$ array (where L reflects the length of the chromosome), which is suggested by the two nested loops, to a single column of depth N. This is as a result of taking a projection of the graph into 1D space. We reduce the array to a column by selecting the j axis as time.

In the cell we use a delay cell to remove the stagger on the chromosomes. This allows Crossover to occur between genes in the same position (locus). Once crossover has been applied, a further delay cell is used to delay the bottom chromosome to restore the stagger.

3.4. Mutation

Mutation is very similar to Crossover. It is a bit-wise operator and employs the same random number scheme as used above although it does not pair chromosomes. Indexing the code is straightforward (Fig 7) and follows the treatment of the Crossover. The only difference is the need to pipeline the Mutation threshold, P_{mut}. This value is used to limit the application of Mutation. We pipeline P_{mut} in the i direction so that it desends the column of the final array. Notice we also pipeline P_{mut} in the j direction but this is removed when we project this dimension out.

```
for(i=0; i<N; i+=2) {
  for(j=0;j<L; j++) {
    if (randval()<pmut[i][j]) {
      chr[i].gene[j]=!chr[i].gene[j];
      } else {
      chr[i].gene[j]=chr[i].gene[j];
    }
    pmut[i][j+1]=pmut[i][j];
  }
  pmut[i+1][0]=pmut[i][0]; }
```

Figure 7. C code for Mutation

4 Conclusions and Further Work

Our design for a hardware genetic algorithm has resulted from the systematic application of systolic array synthesis techniques to the basic algorithm expressed as C code. Considering that the genetic algorithm has dynamic index features it is not an obvious choice for systolic implementation. Our work gives some indication as to the range of algorithms that can be manipulated in this manner. We have also presented a new Select array. This has eliminated the need for three arrays in our previous design.

We have implemented a modest design (N=4) onto an FPGA and can achieve a throughput of over 16 million genes per second. Current work is concentrating on developing the algorithm. By removing the random element from the synthesis process we are able to apply synthesis techniques to new selection and reproduction operators.

References

[1] I. M. Bland and G. M. Megson. Systolic random number generation for genetic algorithms. *Electronic Letters*, Vol. 32(12):1069, 1996.

[2] E. Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, IlliGAL, University of Illinois, Jun. 1995.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[4] G. M. Megson. *An Introduction to Systolic Algorithm Design*. Clarendon Press, Oxford, 1992.

[5] G. M. Megson and I. M. Bland. Generic systolic array for genetic algorithms. *IEE Proc. Computers and Digital Techniques*, 144(2):107–121, Mar. 1997.

[6] S. D. Scott, S. Sharad, and S. Ashok. A hardware engine for genetic algorithms. Technical Report UNL-CSE-97-001, University of Nebraska-Lincon, June 1997.

[7] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 2–9. San Mateo: Morgan Kaufmann, 1989.