Fractal Generation on a Reconfigurable Architecture

Steven A. Guccione

Mario J. Gonzalez

Computer Engineering Research Center

Department of Electrical and Computer Engineering

The University of Texas at Austin

Austin, TX 78712

Email: guccione@cerc.utexas.edu

MGonzalez@utsystem.edu

March 25, 1994

Abstract

Existing FPGA-based reconfigurable machines rely primarily on hardware descriptions to implement algorithms. One proposed alternative is to specify algorithms using a vector-based data-parallel programming methodology. This methodology permits high performance, pipelined circuits to be extracted directly from high-level language descriptions. In this paper, a program and corresponding circuit to generate the Mandelbrot set are discussed. Issues concerning functional decomposition and conditional statements and their corresponding circuits are also explored.

1 Introduction

Several high-performance FPGA-based reconfigurable machines have recently been designed and implemented [1] [3], [4], [8] [10]. These machines are currently programmed at the hardware level. While this approach permits the most flexibility and the highest performance, it requires programmers to be skilled hardware designers.

One proposed approach to programming reconfigurable machines at a higher level is to begin with descriptions of algorithms using a vector-based data-parallel methodology [5]. These descriptions may be translated directly into high performance pipelined circuits. These circuits may then be implemented directly on the reconfigurable machine.

It is expected that computationally intensive algorithms such as those currently executed on supercomputers and large parallel machines will be best suited to this approach. An example application, the generation of a popular type of fractal, the Mandelbrot set, is examined here. This application introduces the use of functional decomposition and a method of performing conditional calculations.

2 The Mandelbrot Set

The algorithm to calculate the Mandelbrot set is a popular computationally intensive algorithm. The ubiquitousness of this algorithm has made it something of a benchmark for high performance systems. Much of the popularity of this algorithm can probably be attributed to the interesting bitmaps it generates. But there are two particular features of the algorithm that make it computationally interesting.

First, it is a compute bound algorithm. No input or output is performed during the bulk of the calculation. Second, the calculation parallelizes easily. This has led to its extensive use as a demonstration vehicle for parallel machines.

The formula for the algorithm is very simple. A single quantity is recursively calculated using the formula:

$$z = z^2 + c$$

In this equation, z and c are both complex numbers. Initially, z is set to zero, and c is set to some initial condition. The value of z is then iteratively calculated. Since this calculation is non-linear, the value of z can be expected to either remain within fixed bounds, or diverge toward infinity.

When calculating the Mandelbrot set, the quantity of interest is not the actual value of z, but rather the number of iterations taken before the equation diverges. It is known that when the magnitude of either the real or the imaginary portion of z becomes greater than 2, the values will diverge toward infinity. The number of iterations taken to reach this point of divergence is the quantity of interest.

In most implementations of the Mandelbrot set algorithm, several initial conditions are calculated together. These points are usually equally spaced within the complex plane. The iterations are then plotted graphically, with individual display pixels corresponding to the points in the complex plane. Since each calculation is independent, there is a large amount of parallelism exploitable in the algorithm.

3 Functional Decomposition

The calculation of the Mandelbrot set makes extensive use of complex arithmetic. In the complex number system, values are represented as pairs of numbers describing the real and imaginary components.

In the vector model of computation, all quantities are linear arrays of contiguous values. To perform calculations using complex values, a complex vector data type must be specified. This data type will consist of two standard vector data types grouped into a *C*-like structure.

The two complex operations used in the calculation, addition and multiplication, must also be specified in terms of existing arithmetic or logical operations. Complex addition is fairly simple. The real and imaginary parts of the vector are added, respectively. The data parallel code for this function is shown in Figure 1. This code uses *C++* style operator overloading.

A digital circuit can be extracted from this function. This circuit is derived from the dataflow graph of the code. Since there are two addition operations, and both are indepen-

```
/* A Complex vector */
struct Complex {
          Vector  Re;
          Vector  Im;
          };

/* Complex addition */
Complex "+"(Complex a, Complex b) {
   Complex sum;

   sum.re = a.re + b.re;
   sum.im = a.im + b.im;

   return (sum);
   };  /* end "+" */
```

Figure 1: The code for complex addition.

dent, the circuit is fairly simple. Figure 2 shows the extracted circuit from this function. Using a standard addition macrocell, a new complex addition macrocell is created. This new cell can be used in conjunction with existing macrocells. It may even be used in the construction of other, more complicated macrocells.

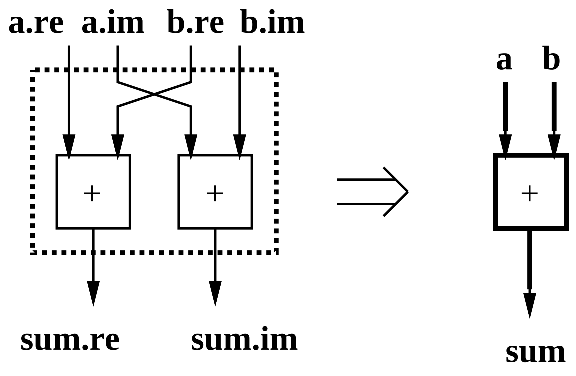

Figure 2: The complex addition circuit.

In a similar fashion, data parallel code can be written to multiply two complex vectors. Note that the operation required is actually a square of the value of z. Rather than implement the special case of a squarer, the more general multiplication unit is implemented. The code used to implement complex multiplication is shown in Figure 3.

As with the addition operation, the dataflow graph of the code may be constructed

```
/* Complex multiplication */
Complex "*"(Complex a, Complex b) {
   Complex prod;

   prod.re = (a.re * b.re) - (a.im * b.im);
   prod.im = (a.im * b.re) + (a.re * b.im);

   return (prod);
   };  /* end "*" */
```

Figure 3: Code for complex multiplication.

and used to produce a digital circuit implementation of the operation. In this case, four multiplications, an addition and a subtraction are performed. The circuit in Figure 4 is extracted from the dataflow graph of the code.

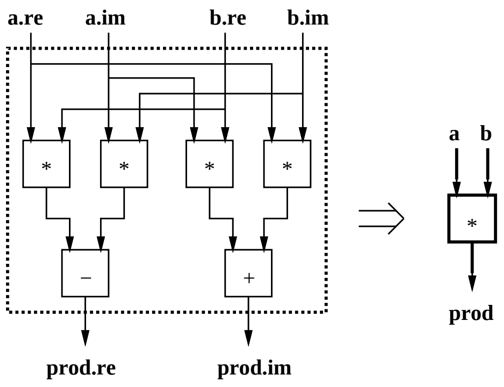

Figure 4: The complex multiplication circuit.

The complex addition and multiplication functions demonstrate that problems can be decomposed into smaller, more manageable units. This technique has been used successfully by both hardware and software designers to manage complexity. These two methods are analogous, as demonstrated by this approach. The reusable software modules are translated into corresponding reusable hardware modules.

4 The Initial Condition Vector

With the necessary support for complex arithmetic in place, a data parallel description of the algorithm can be written. The first issue concerns the values in the initial condition vector, c. This vector will contain the real and imaginary values representing points in the complex plane. The simplest approach to supplying these values would be to consider this vector a static constant that is initialized before calculation begins.

While a simple alternative, these points would still have to be calculated, perhaps off-line, by some host machine. It would be desirable to calculate these values using the reconfigurable hardware.

The values to be generated are points in the complex plane. If the X-axis is considered to represent the real values, and the Y-axis imaginary values, the problem is just one of producing evenly spaced (X, Y) points in this plane.

Since the values used are real and imaginary vectors, data parallel operations on each value in the vector must be used. One approach to producing the desired vectors is:

1. Declare a vector of length $(X\_SIZE * Y\_SIZE)$
2. Generate non-negative integer points in the plane
3. Scale the values
4. Offset to the proper coordinates

The code which uses this approach to generate the c vector is shown in Figure 5. For simplicity, each line of code in Figure 5 performs a single operation. It should be noted that since only two quantities are being calculated, it is possible to merge the 12 lines of code in the example into two somewhat more complex lines of code.

First, in generating the non-negative integer values, the parallel prefix *add-scan()* operation is employed. Along with a constant initialization and a constant subtraction, this operator is used to produce vectors which range from 0 to $(X\_SIZE * Y\_SIZE)$. The modulus operator (%) is used to produce the real portion of the vector, breaking the vector into Y_SIZE segments with values running from 0 to X_SIZE.

In a similar fashion, the integer division operator (/) is used to break the imaginary vector into Y_SIZE segments containing all 0s in the first segment, 1 in the next segment,

```
/* Calculate real portion of vector */
c.re = 1;                   /* [1, 1, 1, 1, ...] */
c.re = add-scan(c.re);      /* [1, 2, 3, 4, ...] */
c.re = c.re - 1;            /* [0, 1, 2, 3, ...] */
c.re = c.re % X_SIZE;       /* [0, 1, 2, 3, ... 99, */
                            /* [0, 1, 2, 3, ... 99, */
                            /* ... */
                            /* [0, 1, 2, 3, ... 99] */

/* Calculate imaginary portion of vector */
c.im = 1;                   /* [1, 1, 1, 1, ...] */
c.im = add-scan(c.im);      /* [1, 2, 3, 4, ...] */
c.im = c.im - 1;            /* [0, 1, 2, 3, ...] */
c.im = c.im / Y_SIZE;       /* [0, 0, 0, 0, ... 0, */
                            /* [1, 1, 1, 1, ... 1, */
                            /* ... */
                            /* [99, 99, 99, ... 99] */

/* Scale real */
c.re = c.re * X_SCALE;      /* [0.00, 0.02, ... 1.98, */
                            /* ... */
                            /* [0.00, 0.02, ... 1.98] */
c.re = c.re + X_START;      /* [-1.00, -0.98, ... 0.98, */
                            /* ... */
                            /* [-1.00, -0.98, ... 0.98] */

/* Scale imaginary */
c.im = c.im * Y_SCALE;      /* [0.00, 0.00, ... 0.00, */
                            /* ... */
                            /* [1.98, 1.98, ... 1.98] */
c.im = c.im + Y_START;      /* [-1.00, -1.00, ... -1.00, */
                            /* ... */
                            /* [0.98, 0.98, ... 0.98] */
```

Figure 5: Code for the initial condition vector.

etc ...

The scaling and translation is accomplished using a constant multiplication and an addition. The technique used is the same for both real and imaginary portions of the c vector.

Employing the functional decomposition technique that was used by the complex arithmetic circuits, a circuit for producing the complex vector c can also be constructed. Figure 6 shows a diagram of this circuit. Note that only constant values are used as circuit inputs. These constants may be integrated directly into the circuit, rather than input as vectors.

Constant folding can be accomplished in one of two ways. First, standard arithmetic

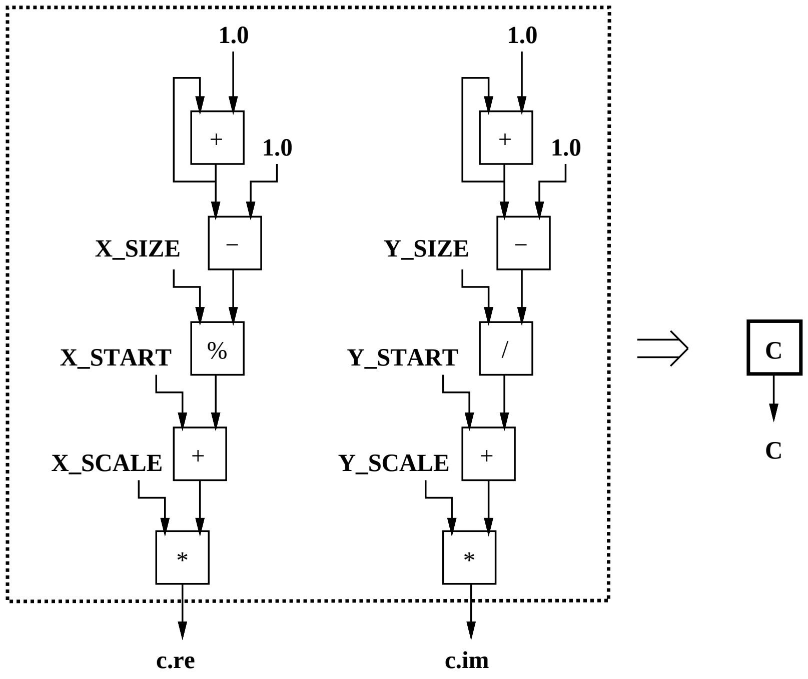

Figure 6: The initial condition circuit.

circuit macrocells such as addition and multiplication may be used, with the constant values placed in a dedicated register at an input of the macrocell. The second alternative is to construct custom optimized macrocells. These custom macrocells can be viewed as optimized versions of the standard macrocell. For example, division of an integer by an even power of two, can be implemented as a simple shift operation. In many cases, the size and speed of these optimized circuits is a considerable improvement over the general case.

As with the complex arithmetic operators, this code may be packaged as a function and used as a macrocell. Note that the *c* macrocell has no inputs but outputs a new complex value with each clock cycle. This lack of inputs reduces the required bandwidth of the final circuit.

5 Conditionals

The Mandelbrot set algorithm is not yet implementable using the existing programming model. Currently, only arithmetic and logical operations are permitted. This calculation requires some conditional calculation. First, diverging values in the computation of z will eventually produce arithmetic overflow. Vector elements larger than some fixed threshold should not participate in the calculation. Also, the incrementing of the pixel values is only performed when the value of the z vector element is below this threshold. What is necessary is programming language support for conditionals and some method for converting these language structures into circuits.

Athanas describes a method for implementing the *IF* statement from the *C* language. The statement is mapped onto multiplexer or MUX based circuits [1]. This approach was used to produce combinational circuits. An extension of this method to the data-parallel programming model will now be demonstrated. This will permit conditional code statements to be translated into pipelined circuits for vector computations.

In the conditional statement in Figure 7, vector x is assigned to some function $f(x)$ when the clause *cond* is *true*. In a traditional instruction set machine, this code represents a control structure which would generate a comparison and a branch instruction. If *cond* is *false*, the assignment statement will simply not be executed.

```
if (<cond>)
   x = f(x);
```

Figure 7: A conditional statement.

Translating this code statement into a digital circuit, particularly a pipelined circuit, presents some problems. First, the dataflow graph representation of the code must be extended to account for the conditionals. In addition, some technique for converting these graph structures to circuits must be specified.

If a MUX-based solution is considered, it is possible to consider the MUX a macrocell, much like the other arithmetic and logical macrocells. The MUX macrocell, however, takes three, rather than two inputs. Two of these inputs will be data inputs corresponding to the *true* and *false* cases. The third input will be a single bit for the select line.

Using this approach, two distinct alternatives must be supplied for each conditional clause. This approach introduced here will be referred to as the *dual assignment rule*. For each value assigned in the *if* clause of a conditional statement, there must also be a corresponding assignment in the *else* clause. These values are calculated in parallel and selected appropriately.

In Figure 7, the lack of an *else* clause implies an identity assignment. The implied code is shown if Figure 8.

```
if (<cond>)
   x = f(x);
   else
      x = x;
```

Figure 8: The implied dual assignment.

This implied assignment supplies the MUX with its two necessary inputs. This permits a valid dataflow graph to be constructed. A circuit for this code fragment is shown in Figure 9.

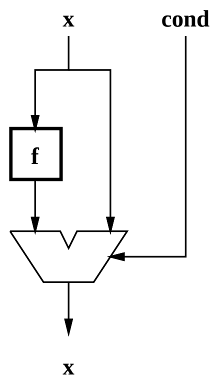

Figure 9: The conditional circuit.

Pipelining this circuit is fairly straightforward. Both the conditional operator and the MUX can be implemented with registered outputs driven by the common system clock. This permits both to be viewed as standard data-parallel vector operations at the software level. The *greater-than* conditional operator, as illustrated in Figure 10 can be viewed as a vector operation.

a:	[0,	-1,	6,	8,	1,	-6]
b:	[4,	3,	-2,	2,	9,	-8]
$a > b$:	[0,	0,	1,	1,	0,	1]

Figure 10: The vector conditional statement.

6 The Calculation

With the code and corresponding circuits for complex operators and conditional statements available, the implementation of the algorithm can proceed. Figure 11 shows the code used to implement the Mandelbrot set.

```
if (z < 4.0) {
   z = (z * z) + c;
   pixel = pixel + 1;
   }
```

Figure 11: The code to calculate the Mandelbrot set.

As in the other examples, this data parallel code is used to produce a pipelined digital circuit. Figure 12 shows the final circuit extracted from the dataflow graph of the code. Note that the previously defined "c" and complex arithmetic operators are used in this circuit.

The final circuit has very low bandwidth requirements. A complex vector z and a vector containing corresponding pixel values is input to the circuit. The updated complex vector z and pixel values are output. A total of three numeric values are input to the circuit and three numeric values output per clock cycle. During this clock cycle, approximately 20 arithmetic operations are performed.

The simulated output of this circuit is reproduced in Figure 13. For monochrome reproduction, the image below was produced by thresholding an 8-bit bitmap. The familiar outline of the Mandelbrot set is clearly visible.

The circuit, as implemented, produces successively better approximations to the actual Mandelbrot set values with each pass. It is assumed that the routing of data through the

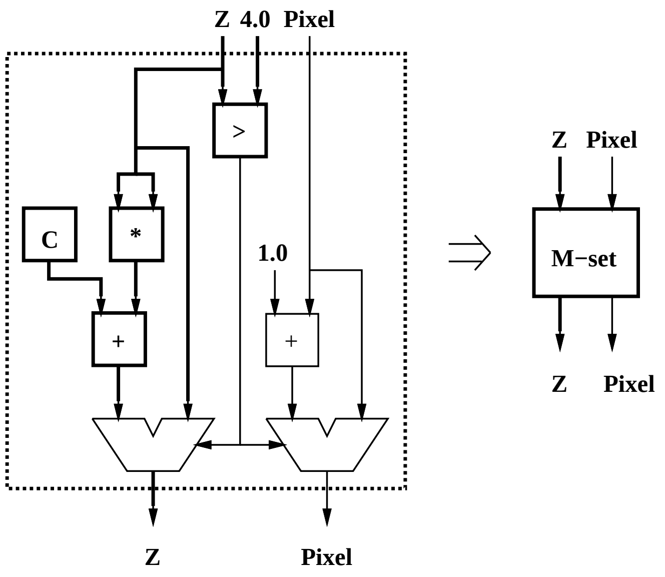

Figure 12: The Mandelbrot circuit.

circuit is under direct control of the memory system, and indirectly, under control of the host.

Recall that the c vector is generated based on constant initializations at the beginning of the code. This constant is fed into an *add-scan()* operator. This *add-scan()* operator contains a feedback loop and, hence, contains some state information from the previous pass. This circuit should be reinitialized at the beginning of each pass. This may be accomplished using a reset signal or via reconfiguration. In this situation, partial reconfiguration is particularly helpful.

7 Circuit Complexity and Performance

The circuit extracted from the data parallel code uses 23 macrocells. It is assumed that the complex number *greater-than* operator ($>$) uses three macrocells, two comparators and a boolean AND. The table in Figure 14 gives list of these macrocells. Since a majority of these macrocells take constant inputs, they are tabulated using two values. The first value gives the number of standard macrocells used, the second, the number of macrocells with

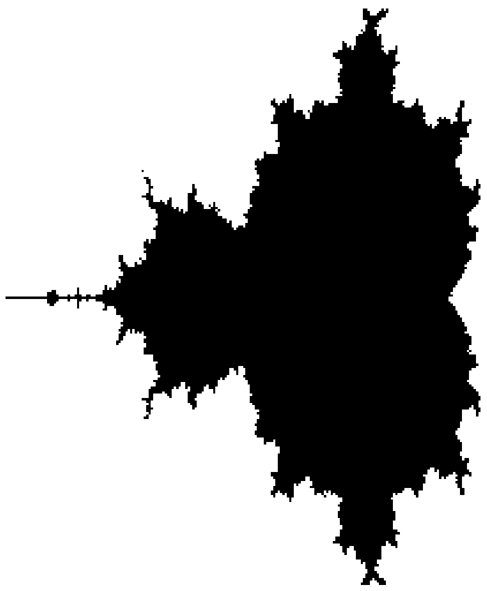
Figure 13: The Mandelbrot set.

constant inputs.

N^2			N			
$*$	$/$	$\%$	$+$	$-$	$>$	MUX
4/2	0/1	0/1	3/5	1/2	0/2	2/0
4/4			6/9			

Figure 14: The macrocell usage.

Macrocells with constant inputs, particularly larger macrocells such as multiplication and division, can benefit greatly from constant folding optimizations. For this reason, the complexity of the circuit is calculated first as a lower bound, representing optimization of these cells, as well as an upper bound, where only standard macrocells are used.

For multiplication and division, a complexity of N^2 is assigned. Optimization is assumed to reduce this complexity to N. All other macrocells are assigned a complexity of N. This complexity measure corresponds roughly to the number of FPGA Configurable Logic Blocks (CLBs) used to implement the function. Other circuit components, such as delay elements for pipeline balancing are neglected in this analysis.

From this table, we calculate the following minimum and maximum circuit complexity:

$$\text{Maximum:} \quad 8N^2 + 15N$$

$$\text{Minimum:} \quad 4N^2 + 19N$$

If the circuit is constructed using all 16-bit datapaths, approximately 1328 CLBs are needed for the minimum estimate and 2288 CLBs for the maximum. While these estimates are rough and depend on the particular FPGA devices used, the number of CLBs necessary is achievable with a very small number of available parts.

As with all pipelined circuits constructed in this manner, one complex result is produced per clock cycle, once the pipeline is filled. In this circuit, the pipeline latency is 7 stages. If the multiplier units are internally pipelined, the latency could increase to several times this value. This is still small, however, compared to the length of the vectors processed.

Given the clock speed, it is simple to calculate the number of pixels processed. The number of pixels processed per second is approximately equal to the clock speed of the circuit. For instance, at 10 MHZ, approximately 10 million pixels are processed per second.

8 Conclusions

The calculation of the Mandelbrot set, a popular computationally intensive algorithm, has been shown to be easily implementatble on a reconfigurable machine. Using a data-parallel programming methodology, very high level code can be used to describe the algorithm. This code can be translated into a high-performance pipelined circuit and programmed into an FPGA-based machine using a reasonable number of CLBs.

The implementation of this algorithm has introduced the use of functional decomposition. This has long been used to manage the complexity of software and circuit design. Here, it is used to manage the complexity of both the software and the circuits it produces.

Finally, a technique for performing conditional vector operations has been presented. This technique performs two alternative calculations in parallel and selects the desired value using a multiplexer.

References

[1] Peter M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration.* Technical Report LEMS-101, Brown University, Division of Engineering, February 1992.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, MA, 1990.

[3] Patrice Bertin, Didier Roncin, and Jean Vuillemin. *Introduction to Programmable Active Memories.* Technical Report 3, DEC Paris Research Laboratory, 1989.

[4] Maya Gokhale, William Holmes, Andrew Kosper, Dick Kunze, Dan Lopresti, Sara Lucas, Ronald Minnich, and Peter Olsen. SPLASH: a reconfigurable linear logic array. In *International Conference on Parallel Processing*, pages I–526–I–532, 1990.

[5] Steven A. Guccione, and Mario J. Gonzalez. A Data-Parallel Programming Model for Reconfigurable Architectures. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, 1993.

[6] Philip J. Hatcher and Michael J. Quinn. *Data–Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, MA, 1991.

[7] W. Daniel Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[8] T. A. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation.* PhD thesis, University of Edinburgh, Department of Computer Science, January 1989.

[9] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. In *Proceedings of the International Conference on Parallel Processing*, pages 180–185, August 1985.

[10] Jouko Viitanen, Tapio Korpiharju, and Hannu Kiminkinen. Mapping algorithms onto the TUT cellular array processor. In *International Conference on Application Specific Array Processors*, 1990.