

Visualization assisted hardware configuration  
for Bioinformatics algorithms

Priyesh Dixit

Advisors:

Drs. K. R. Subramanian, A. Mukherjee, A. Ravindran

May 2005

## Special Thanks

I would like to thank Dr. Subramanian for his faith in my abilities and for selecting me to do this project. I also thank Dr. Mukherjee and Dr. Ravindran for their patience and support throughout the year and making this project possible. Thanks to Josh Foster and Nalin Subramanian for help with the FLTK and VTK. And also I would like to especially thank Jong-Ho Byun for his all his help.

# Table of Contents

<b>Chapter 1: Introduction</b> .....	<b>4</b>
<b>Chapter 2: Background</b> .....	<b>6</b>
2.1 Bioinformatics.....	6
2.2 Field Programmable Gate Arrays.....	6
2.3 The Smith-Waterman algorithm.....	9
2.4 Visualization.....	10
<b>Chapter 3: System Description</b> .....	<b>11</b>
3.1 Overview.....	11
3.2 Design flow.....	12
3.3 Data files.....	13
3.2 Visualization Design.....	16
<b>Chapter 4: Implementation</b> .....	<b>21</b>
4.1 Class Heirarchy.....	21
4.2 Metrics Implemented.....	22
4.3 The Visualization Pipeline.....	23
4.3 User Session.....	24
<b>Chapter 5: Experiments</b> .....	<b>26</b>
5.1 Generating the data.....	26
5.2 Comparing datasets.....	27
<b>Chapter 6: Conclusion</b> .....	<b>30</b>
<b>References</b> .....	<b>31</b>
<b>Appendix</b> .....	<b>32</b>
Application source code.....	32

## Chapter 1: Introduction

Bioinformatics is the development of efficient computational algorithms, statistical analysis methods, and mathematical models, to manipulate, analyze, interpret and distribute biological information in gene and protein databases. The main challenge lies in analyzing the phenomenally enormous amount of accumulated data, and hence, the need for high performance computing. The current computational platforms, such as microprocessor based super computers and grid computing networks, are not fast enough to keep up with increasing complexity of the data [7]. High performance reconfigurable hardware such as Field Programmable Gate Arrays (FPGA's) can fulfill this need.

The proposed research will make it feasible to execute bioinformatics applications and algorithms on compact and cheap computation platforms that are hardware reconfigurable, and orders of magnitude faster than supercomputers. The FPGA based platforms will accelerate the computationally intensive cores of the algorithms, and will be part of standard computation platforms like supercomputers, clusters of PCs (personal computers) and workstations, and grid computers. Beyond the domain of hardware implementation, we are proposing an entire methodology for optimal design of bioinformatics algorithms for efficient hardware implementation, along with a partitioner for efficiently implementing enormous computations on multiple FPGAs, memories and processors. We expect that our research will lead to novel system-level architectural innovations in FPGAs and hypercomputing platforms for bioinformatics applications. The objective of my senior project was to develop a visualization application for this hypercomputing platform.

Visualization is needed for our system in order to visually observe the effectiveness of a proposed hardware mapping. Currently, there is no existing software to our knowledge

that allows the user to visually simulate the execution of an FPGA. The Xilinx [6] ModelSim application allows the user to see the activity in the wires in the form of a wave (whether the wire is high/low). However, the wave form is not easy to understand, especially over time for long running algorithms. We need to explore the many cost and performance metrics of an FPGA over time. Thus, this data is multivariate and visualization can aid in understanding this data. In particular, we use multiple linked views to bring out the spatio-temporal features of interest.

In this project, we have developed a visualization application that enables the user to visualize this multivariate data. The application was developed using the Fast and Light ToolKit (FLTK) [1], the Visualization ToolKit (VTK) [2] and OpenGL (Open Graphics Library). FLTK was used to develop the Graphical User Interface of the application. VTK and OpenGL were used to render the visualizations in the different views. All of the tools used to implement the tool are cross-platform, which means we can port it to almost any operating system. It has three different visualization displays, the main view, the detailed view and the graph view. The system requires two types of data files as input, an architecture description file and a chip activity data file. Once the data has been loaded, one can play the simulation using the VCR controls in the control panel.

The work that has been completed so far is enough to show the potential of our proposed system. Using the visualization, scientists from many disciplines can observe and optimize hardware implementations for faster and more efficient computations. This report will discuss the work that has been done within the past year. It is broken down into several sections. The first section gives some background information on bioinformatics and FPGAs. Next, there is a description of the visualization system and how it was implemented. This is followed by the experiments performed with the tool and their results. The possible future work is outlined in the concluding chapter. The source code of the visualization application is also provided in the appendix.

## **Chapter 2: Background**

### ***2.1 Bioinformatics***

Bioinformatics is the management and analysis of biological (especially genomic) data using advanced computing techniques [3]. The goal of bioinformatics is to solve the many mysteries of science using high performance computational platforms. Currently, most bioinformatics algorithms are performed on multi-processor super computers and distributed computing networks. These platforms are relatively slow and the growth of processing power is not high enough to keep up with the demands of bioinformatics. One example of this would be DNA sequence matching algorithms. There are many different types of sequence matching algorithms such as BLAST, FASTA, HMMER and Smith-Waterman. It involves the comparison of a DNA string of arbitrary length with a gene database (such as GENBANK). Such an algorithm requires a lot of computational power and can take hours on a microprocessor system [7].

### ***2.2 Field Programmable Gate Arrays***

The power required to execute bioinformatics algorithms can be found in high performance Field Programmable Gate Arrays (called FPGAs). An FPGA is an array of logic gates that can be hardware-programmed to execute user-defined tasks. Due to its reconfigurable nature, it can be optimized to perform a task at much greater speeds than regular microprocessors [4]. Application specific integrated circuits (ASICs) on the other hand, are faster and smaller than FPGAs, and dissipate less power compared to FPGAs [8]. However, ASICs are not reconfigurable once fabricated, and hence, they are extremely expensive and have months of turn-around time.

An FPGA contains many computational elements called “Slices” that have some input and output wires that are known as “Nets”. Each net has an origin slice and one or

more destination slices. Modern FPGAs are high performance, capable of massively parallel computations, and complex enough to implement system-on-chip at low costs [11]. Therefore, we propose the use of FPGAs along with standard computing platforms to execute bioinformatics algorithms. This is the concept of a “hyper-computer.” In order to optimize the mapping and partitioning of the algorithms to the hardware, a visualization tool is required. A visualization of metrics such as power, latency and transition density can aid in the optimization of the hardware configuration.

With the help of the visualization, bioinformatics researchers can create their own hyper-computing platforms. First, one must write the algorithm to be mapped in C source code and convert it to the circuit description language VHDL using the ImpulseC software. Next, using the Xilinx software one can map the VHDL code onto one or more FPGA designs. Then, the data files for the visualization would be generated and input into the visualization system. Once the mapping has been optimized, it can be used to generate bit-streams that are sent to the hardware. The reconfigurable hardware platform is a BenNUEY motherboard connected to the host computer via a PCI slot. The motherboard interfaces between the host computer and the multiple FPGAs. There can also be more than one motherboard. The relatively slow PCI bus is only used when communicating with the CPU for downloading bit streams onto the FPGAs, for driving the FPGA inputs and analyzing their outputs, and for inter-motherboard communications. A diagram of the flow is given below.

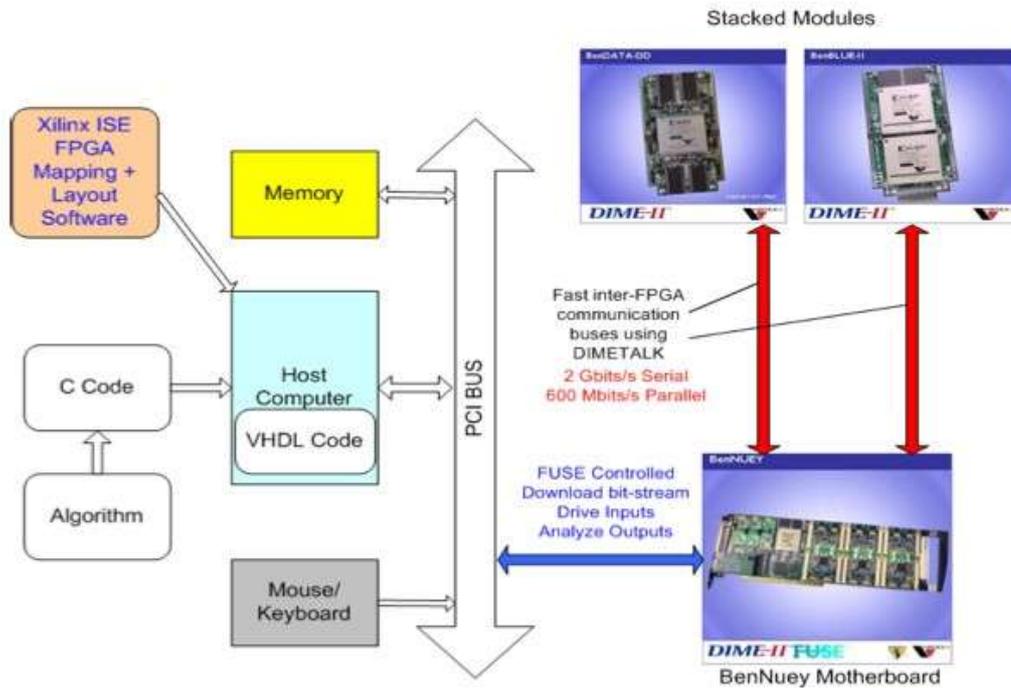


Figure 2.1: Data flow diagram of a typical hyper-computing platform

There are several metrics that must be considered when optimizing the hardware configuration. These include utilization, power, latency, and transition density. They are derived as follows:

- **Utilization:** percentage of chip area being used by the algorithm. This metric is constant for any given design and is the only time-independent variable. The formula is as follows:

$$U = (\text{Used chip area} / \text{Total chip area})$$

- **Transition Density:** the average of the transitions in the input and output nets of a slice during a given time frame. The transition density of a wire is as follows:

$$TD(t) = \text{Total transitions} / \text{Maximum transitions possible}$$

- **Power:** the product of transition density, relative length and the Capacitance Coefficient of the FPGA. Relative length is the length of the net divided by maximum length of the design. The Capacitance Coefficient (Kc) is a hardware

dependent constant. The formula is as follows:

$$P(t) = TD(t) * (L/L_{max}) * K_c$$

- **Latency:** the ceiling of the product of transition density, relative length and the Delay Coefficient ( $K_d$ ), divided by the clock period,  $T$  (in seconds).  $K_d$  is another hardware-dependent constant like  $K_c$ . The formula is as follows:

$$L(t) = \text{ceil}[TD(t) * (L/L_{max}) * K_d] / T$$

### **2.3 The Smith-Waterman algorithm**

For this project the bioinformatics algorithm chosen for our experiments was the Smith-Waterman sequence-matching algorithm [9]. The goal of sequence matching is to determine the similarities between the subsequences of strings as much as possible. The similarity is determined by the minimum changes needed to get one string from the other. A scoring scheme must be used to assign a certain score to a pair of strings based on gaps, matches and mismatches. A matrix is used to keep track of the scores and eventually compute the result, where  $M[i,j]$  is the score for strings  $A[0..i]$  and  $B[0..j]$ . The position  $M[0,0]$  is assigned the value of zero (for two empty strings) and then the rest of the positions are computed in a pipelined fashion.

This algorithm was already implemented in C, it was then converted to VHDL using ImpulseC and then mapped to an FPGA using the Xilinx Integrated Software Environment. The mapping of the algorithm to hardware is also a pipeline. Individual processing elements (consisting of multiple slices) represent a column in the matrix. The computation flows from one processing element to another and finally reaches a counter that calculates the result. Using multiple FPGAs in parallel to compute different sub matrices can optimize this design further. The pipelined nature of the algorithm made it easier to tell if the transitions were displayed accurately.

## **2.4 Visualization**

Visualization is the technique of transforming complex, multivariate data into interactive, sensory representations using a computer in order to better understand the information [5]. The representation is typically visual although it is possible to use other senses such as hearing and touch. In our case, a hardware device is represented on the computer as a two-dimensional grid and attributes such as power are mapped onto the grid as colors. This allows the user to intuitively interpret the complex relationships between points in the data.

Modern visualization tools provide an effective means for extracting knowledge from large amounts of raw or processed data. For the past two decades, visualization techniques have been increasingly employed in a number of scientific, engineering and business domains. Interactive visualization techniques have the added advantage of placing the user expertise in the loop, leading to exploratory visualization environments that can be considered to be an effective search technique for near-optimal solutions. Since our design methodology and hypercomputing platform will be mainly used by biologists and bioinformaticists, visualization will be extremely helpful for them to comprehend the system, and analyze/interpret the observed data. Thus, the users can effectively use our system, and find good partitioning results for high speed sequence matching.

There are several tools available for data visualization such as Prefuse, AVS, IBM Data Explorer and VTK. The tool we chose for this project was the Visualization ToolKit (VTK), which is a C++ class library based on OpenGL. VTK allows for the creation of a pipeline of data sources and filters. It uses a “lazy update” method, which means it does not redraw the visualization unless it something is modified in the pipeline. This makes it much faster and efficient. A diagram of the VTK pipeline used in this application is given in Figure 4.2.

## Chapter 3: System Description

### 3.1 Overview

The interactive visualization system is an intuitive tool that allows the user to monitor the execution of an algorithm on an FPGA board over time. The VCR-like interface allows for simple controls for viewing any part of the simulation. There are three variables that can be adjusted through the user interface: play position, window size and time delay. *Play position* is simply the current time step being displayed on the screen. *Window size* determines the number of time steps that are averaged to calculate the color map value. A moving window average is used, so if N is the window size then the metric at the current time step is added to the N-1 time steps after it and averaged to return the final metric value (see Figure 3.1). The *time delay* is the time (in seconds) between frames during playback. Note that this does not include any delay caused by computations. A picture of the system interface is shown in Figure 3.2.

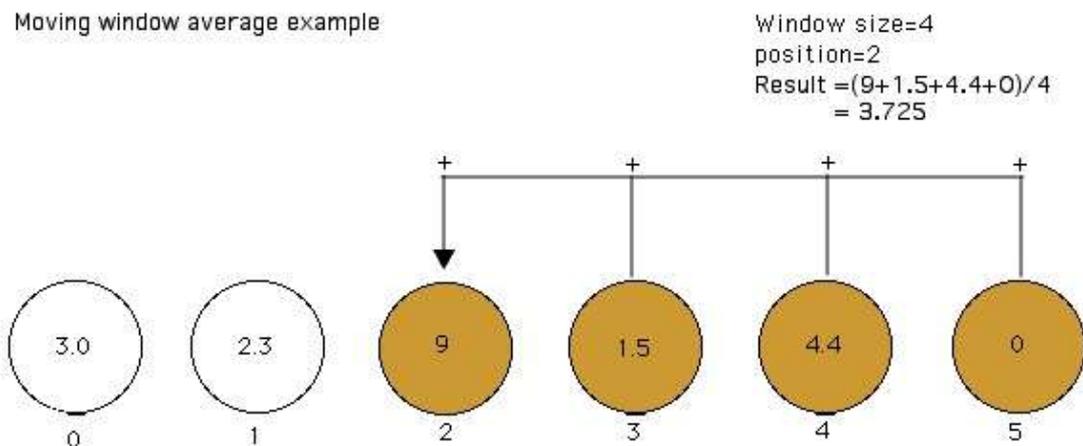
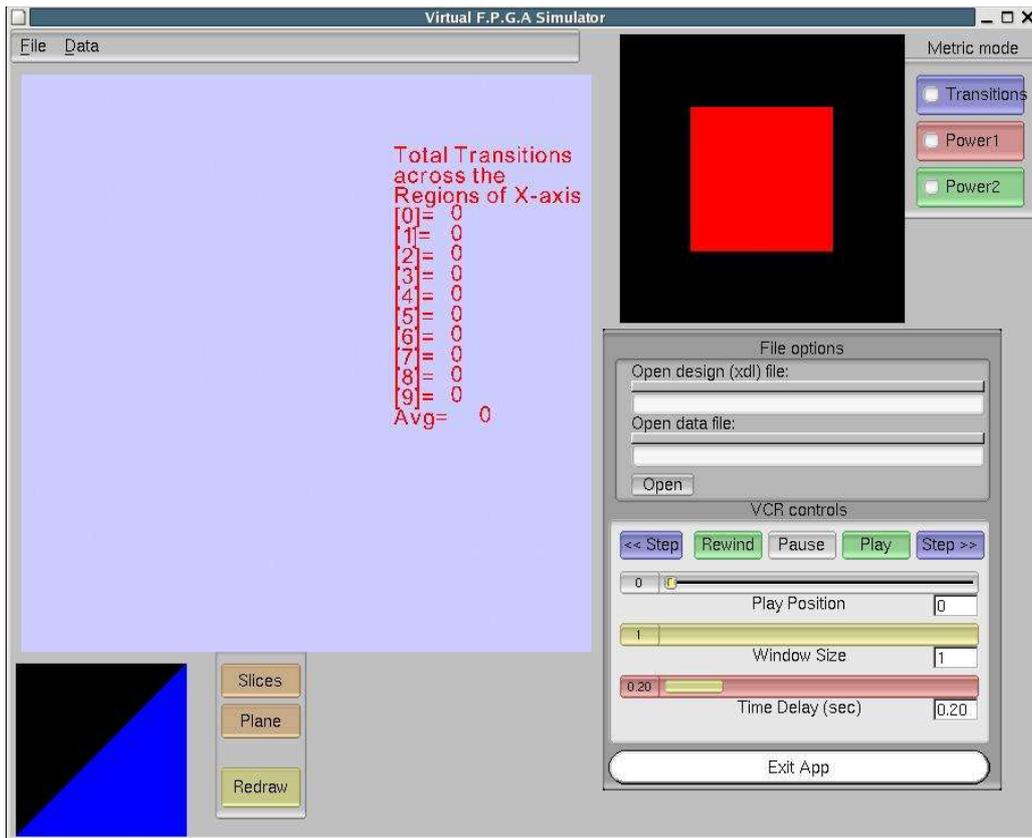


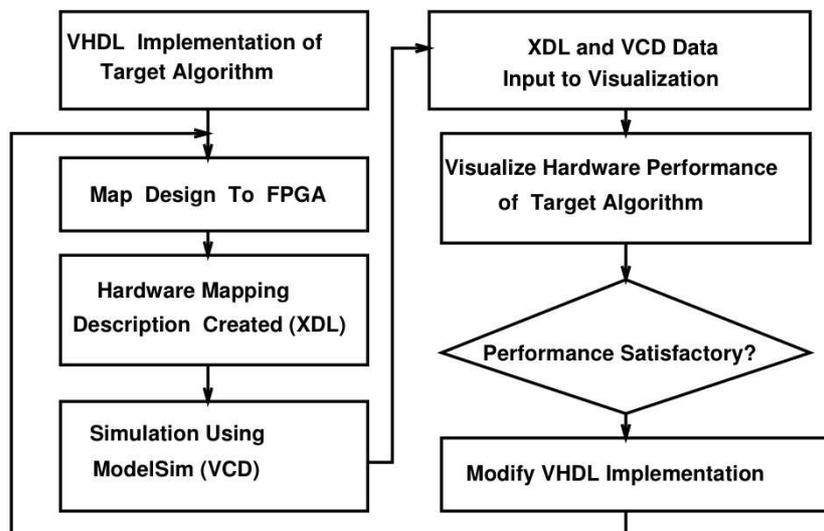
Figure 3.1: Moving window average



**Figure 3.2: Screenshot of the main screen before any data is loaded**

### 3.2 Design flow

The diagram below describes the typical data flow of the system.



**Figure 3.3: Typical design flow of the system**

We start with the VHDL implementation of the algorithm, which then gets mapped to an FPGA. VHDL is a design description language used to map algorithms onto FPGAs and other hardware devices. Next, we generate a data file that contains the hardware mapping description of the chip (XDL). After the XDL file is generated, the simulation must be done using the Xilinx ModelSim simulation software and a VCD file is created. These two files are input into the visualization tool and the VHDL implementation can be modified based on the performance. The data files are explained in more detail in the following section.

### 3.3 Data files

There are two different types of data files that can be loaded into the visualization. The first is the XDL (Xilinx Design Language) file that shows the architecture of the design. The XDL file contains a list of all slices in the circuit along with their X,Y coordinates. The file also lists all nets along with their origin slice and destination slices. Once this file is parsed another file containing the activity information must be loaded. A sample of an XDL file description of a slice and a net is given below.

```
inst "Tout_1_OBUF" "SLICE" , placed R91C31 SLICE_X61Y10 ,
  cfg "CYSELF::#OFF CYSELG::#OFF COUTUSED::#OFF YUSED::#OFF
    XUSED::#OFF F5USED::#OFF YBMUX::#OFF CYINIT::#OFF DYMUX::0
    DXMUX::#OFF F::#OFF: REVUSED::#OFF FXMUX::#OFF
    SYNC_ATTR::ASYNC SRFFMUX::#OFF FFY_SR_ATTR::SRLOW FFX::#OFF
    FFY:y8/output:#FF FFX_SR_ATTR::#OFF G_ATTR::#OFF
    DIG_MUX::#OFF CY0G::#OFF GYMUX::#OFF FXUSED::#OFF
    DIF_MUX::#OFF SLICEWE0USED::#OFF F_ATTR::#OFF CY0F::#OFF
    DIGUSED::#OFF SHIFTOUTUSED::#OFF BYOUTUSED::#OFF
    FFX_INIT_ATTR::#OFF FFY_INIT_ATTR::INIT0 G::#OFF:
    XBMUX::#OFF BYINVOUTUSED::#OFF SLICEWE2USED::#OFF
    SLICEWE1USED::#OFF WF1USED::#OFF WF2USED::#OFF
    WF3USED::#OFF WF4USED::#OFF WG1USED::#OFF WG2USED::#OFF
    WG3USED::#OFF WG4USED::#OFF YBUSED::#OFF BXOUTUSED::#OFF
    BYINV::BY BXINV::#OFF CEINV::#OFF CLKININV::CLK SRINV::#OFF
    SOPOUTUSED::#OFF SOPEXTSEL::#OFF"
```

...

```
net "Tout_1_OBUF" ,
  outpin "Tout_1_OBUF"   YQ      ,
  inpin  "Tout<1>"      O1      ,
```

```

pip R91C31 YQ2 -> OMUX0 ,
pip R92C31 OMUX_S0 -> S2BEG0 ,
pip R95C31 S2END_S0 -> S2BEG8 ,
pip BIOIC31 S2END8 -> IOIS_FAN_BX3 ,
pip BIOIC31 IOIS_FAN_BX3 -> IOIS_FAN_BY0 ,
pip BIOIC31 IOIS_FAN_BY0 -> O1_B0 ,
# net "Tout_1_OBUF" loads=1 drivers=1 pips=6 rtpips=0

```

Above is an excerpt from an XDL file describing the slice Tout\_1\_OBUF and also a net of the same name. As can be seen, the net comes out of the Tout\_1\_OBUF (outpin) and goes into Tout<1> (the second bit of the Tout line). The other information, such as internal ports and interconnection points, is not used at this time.

The main file type for activity data is the VCD file, which is generated by the Xilinx simulation program called ModelSim. The VCD file is a text file that contains a list of all wires and at each time step it lists the wires that changed in value (had a transition). The VCD file first lists all wires by name with their assigned ID (one or more characters). After the wires have been listed with IDs, a time step is started with a pound sign (#). Following the time step line, the transitions are defined by a 0 or 1 followed by the wire ID. This information is stored in the application such that, for each wire there is an array of time steps when the wire changed in value. The actual value of the wire is currently not recorded. A sample VCD file format is shown below.

```

$date
  Thu Dec 9 21:55:27 2004
$end
$version
  ModelSim Version 5.7c
$end
$timescale
  1fs
$end
$scope module tb_sw $end
$var wire 1 ! clk $end
$var wire 1 " init $end
$var wire 1 # initout $end
$var wire 1 $ tbase_in [1] $end
$var wire 1 % tbase_in [0] $end
$var wire 1 & edit_dist [6] $end
$var wire 1 ' edit_dist [5] $end
$var wire 1 ( edit_dist [4] $end

```

```
$var wire 1 ) edit_dist [3] $end
$var wire 1 * edit_dist [2] $end
$var wire 1 + edit_dist [1] $end
$var wire 1 , edit_dist [0] $end
$upscope $end
$enddefinitions $end
```

...

```
#100000000
1!
0"
0%
1$
```

The above VCD file sample shows the declaration of wires and the symbolic ID's of the wires that are used to reduce file-size. After the wire definitions, I have shown the transitions for time step 100000000 femto-seconds. It shows that wire ! (clk) became one, wire “ (init) became zero, wire % (tbase\_in[0]) became zero and wire \$ (tbase\_in [1]) became one.

When an XDL file is selected, the VTK canvas displays the architecture of the FPGA design. The unused positions in the 200x200 grid are blue and slice positions are green. A data file must be loaded next; it can be either in VCD, compressed (cmp), or binary (bin) format. It must first be generated in VCD format by the Xilinx ModelSim application. The procedure for generating the VCD file is explained in Chapter 5. Once the VCD file is opened it can then be converted to a more compact format by the visualization tool. Be aware that an XDL file must be loaded before the data file. When converting to another file format, the application only conserves the nets that were present in both the XDL file and the original data file. Once the data file is loaded, the metrics can then be calculated.

There are two compressed file formats that were developed for converting the VCD files. The first is a compressed text file (.cmp) that contains only the wires of interest and lists the index of the time steps at which a transition occurs in each wire. It first gives the total number of wires, then the total number of time steps. After that, it has

the name of the first wire followed by the total number of transitions of the wire. It then lists the time step indices of the wire which are then stored in memory. This facilitates for easy file parsing since the data is in the same structure as it is going to be stored in memory. There are two files that are stored for the CMP format, the actual transition data is in the .cmp file and a separate .cmp.steps file is created that contains the absolute time steps rather than indices.

The second format is a binary file (.bin) that contains the exact same information as the .cmp but in an even more compressed form. First it lists the number of wires and the number of time steps. This is followed by the number of transitions of the first wire, and then the time step indices. For simplicity, the names of the wires are not stored in binary form but instead a separate text file with a .bin.names stores an array of the names and it is read concurrently with the binary file. After the transition time steps of the first wire, the number of transitions in the next wire is given, this corresponds to the next wire name in the names file. The VCD file can be converted using the *convert* command line application found in the “binary” directory of the source tree.

### **3.2 Visualization Design**

There are 3 different displays for the visualization. The first is the VtkCanvas, this is the main canvas for showing the visualization. This canvas uses the Visualization Tool Kit (VTK) for rendering. VTK is a widely used tool for visualization applications and is based on OpenGL (Open Graphics Library). It displays the 2D grid of points representing an FPGA either as a single textured plane or a 2D array of glyphs. Currently, the grid size is set to a constant of 200x200. The single textured plane view renders faster but does not have any interactive picking. The glyph view (or Slice View) does allow for picking but due to the large number of objects it takes longer to render.

There are currently two interaction modes in the VtkCanvas that can be chosen from the menu bar. The first is the Trackball mode that allows for zooming and panning of the view with the mouse. The second is the Pick Mode that only works in Slice View. While in Slice View, if the Pick Mode is active, the user can right click on a point to pick a slice at that location. A region of interest can also be selected by dragging the middle mouse button from one point to another. When a slice or range is selected, it changes the display on the GLCanvas (discussed below). The VtkCanvas also shows some statistics in text next to the simulation. This is the histogram of transition density across the regions of the X-axis (broken down into 10 regions horizontally). However, this information is again calculated on the fly separately.

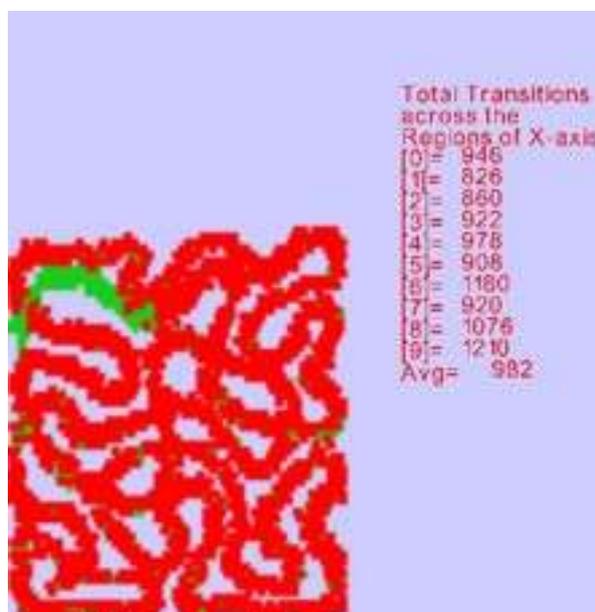


Figure 3.4: VtkCanvas display

The upper-right view in Figure 3.2 is the “detail” view and can show selected parts of the main view in more detail. For instance, the processing elements and their connections. This view uses pure OpenGL to render the slices as rectangles and it also renders the wires connected to the selected slice as lines. When a slice is selected, it is displayed in red, the input slices are yellow and the output slices are blue. The input and output nets are also shown as yellow or blue lines respectively. When the Slice view is active, one can select an area of the chip to view by dragging the middle mouse

button from one corner of the desired view to another. This will show only that area in the detailed view. Interactive picking is possible in the detailed view by clicking the middle mouse button. The view can also be panned by dragging the left mouse button.

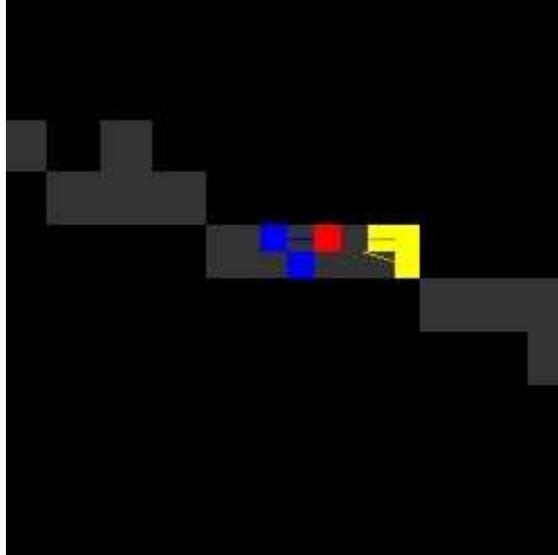


Figure 3.5: GLCanvas/Detailed view shows wires and slices.

The third view window in the bottom left corner is the graph view. There are currently two metrics that can be displayed here, transition density across the regions of the X-axis and net length across the same regions. In transitions mode, transition density is displayed otherwise if we are in Power mode then net lengths within the power range are displayed. The net lengths display is constant for any given circuit. The transition density, however, changes at each time step and is automatically updated. This process is currently very inefficient because the graph view has to iterate through all the slices of the circuit independently of the main view, which does the same thing. In future revisions, the metrics will be calculated once and stored in the application and not recalculated at each time step on the fly.

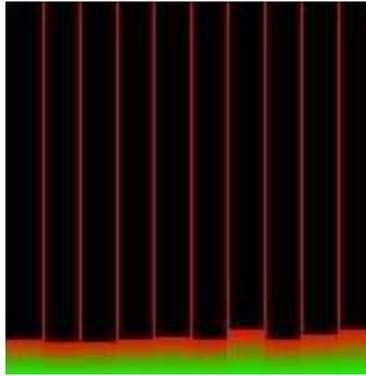


Figure 3.6: Graph view shows a histogram of transition density across the X-axis

All the controls for the application are available through the control panel. The control panel allows the user to open data files by typing the filename and path. In the future this will be improved by having a file chooser. Also, all of the VCR controls for the simulation are available here. The current time step, window size for averaging and speed of playback can also be set using slider bars. There are buttons on the main window to change the metric displayed on the VtkCanvas and Graph view. An image of the control panel is given below.

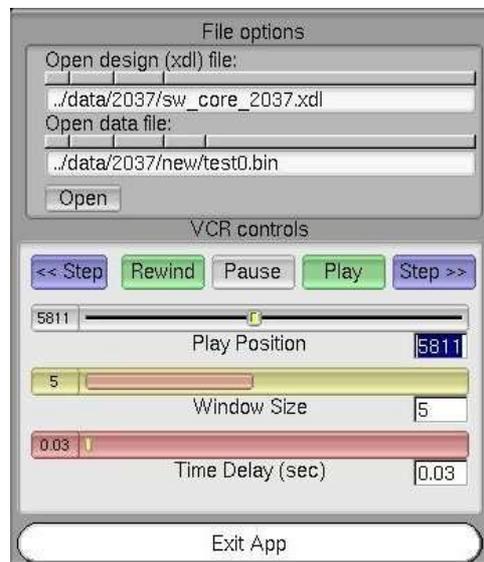


Figure 3.7: Control panel of the application

The menu bar on the main window also contains some features. One can toggle image

capture mode on and off. With image capture mode, an image of the VtkCanvas is saved to a preset location every time the view is rendered. Currently, the save location is “pictures/image-X.jpg” where X is the index of the picture in the sequence. Currently, it skips two frames before saving a frame to conserve memory. Another feature that is available through the menu is the Interaction mode, which was discussed earlier, and the option to save to another file format.

## Chapter 4: Implementation

### 4.1 Class Heirarchy

The architecture of the application is very object oriented. It is written completely in C++ and the Fast and Light TookKit (FLTK) was used to develop the graphical user interface and the FIVtkRenderWindowInteractor was used to combine FLTK and VTK in the same window [10]. A diagram of the classes and dependencies is given below.

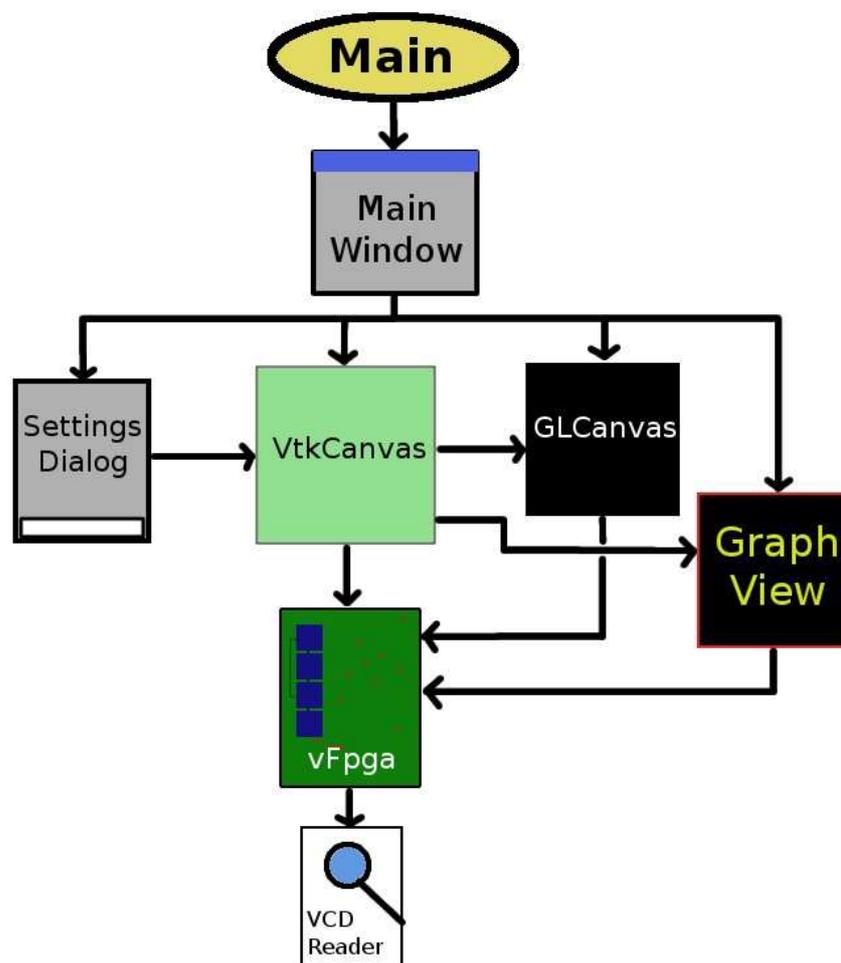


Figure 4.1: Class Diagram, interconnections between the main classes of the application

The **vFpga** class stores all the information about the architecture of the FPGA and the activity data that is parsed by the **VCDReader** class. The different views have pointers

to the instance of the vFpga in order to calculate metrics on the fly. Once the XDL and data file have both been loaded, the information is stored in vFpga. The architecture information is in the form of a hash-map that stores the slices by their ID. The ID of a slice is computed as  $Y * \text{rowlength} + X$ , where  $(X,Y)$  is the coordinate of the slice. The nets are also stored in a hash-map, but the key is the name of the net. Storing the nets by name allows for quick access when we are traversing the FPGA because we store the names of the incoming and outgoing nets of a slice.

The SettingsDialog class is the control panel and does all of the playback logic. When the user selects a VCR action (Play, Rewind, etc) the VCR state is updated and the animation callback function is invoked. It checks the current state and does the appropriate action, and changes the state if needed. The VtkCanvas handles the visualization pipeline but it also controls the Detailed View, the Graph View and the vFpga class. The visualization pipeline is described in detail in section 4.3.

## ***4.2 Metrics Implemented***

Three metrics have been implemented: Transition Density, Power and Latency. The time frame for calculating transition density is determined by the current time step plus the window size (selected in the control panel). The transitions of all wires in a slice are summed up and normalized before being mapped to a color.

The Capacitance coefficient, Delay coefficient and Clock Period are all defined as 1.0 at this time. This means that the output for Latency will be equal to the Power, this is only until we get the definite values for the constants. As you can see, transition density is the main metric that the others depend on. It is also the only time dependent variable in the equations. Length is dependent on the given wire and Max Length is a constant for any given architecture.

Currently it is not possible to obtain the exact net length, so an approximation is done. Since the Latency gives the same result as Power, I implemented Power1 and Power2 buttons instead. Power1 will display power metrics for only the nets whose lengths are in the smaller half of all nets. Power2 on the other hand, displays power for only the bigger half. This allows the user to view power in more detail for the smaller and larger nets.

### **4.3 The Visualization Pipeline**

The VTK pipeline of the application uses the StructuredPoints data type to store a grid of points representing the FPGA in main memory. Each point in the grid has some scalar value and vector value. The scalar value is the current metric value of that position and determines the color in the visualization. A vector is placed at each grid position and the components are set to two if a slice exists and zero if it is empty. The vector value is then used to determine whether a glyph is drawn at that position and also as the scale factor for the glyph. For the textured plane view, the `vtkStructuredPoints` is passed through an image filter to convert it into a texture. That texture is then applied to a plane that is rendered by VTK. In the glyph view, the `vtkStructuredPoints` is passed through the `vtkGlyph3d` filter. There are 2 glyphs that are indexed 0 and 1, the first is a plane and the second is null (no slice). The scalar values are used to color the glyph and the vector is used as the index of the glyph. The glyph view is substantially slower due to the number of planes that must be drawn, compared to only one plane with the texture view. A color lookup table is created with four color values (dark blue, green, light red, bright red). Depending on the view mode, either the glyph or plane geometry is passed to the `PolyDataMapper`. The mapper is then assigned to an actor that is passed to the `Renderer`. If the texture view is active, the texture is applied to the actor. The `Renderer` is passed to the `RenderWindow`, which is in turn passed to the `RenderWindowInteractor`. The `RenderWindowInteractor` is then started and it allows for interaction with the window. A diagram of the pipeline is given below.

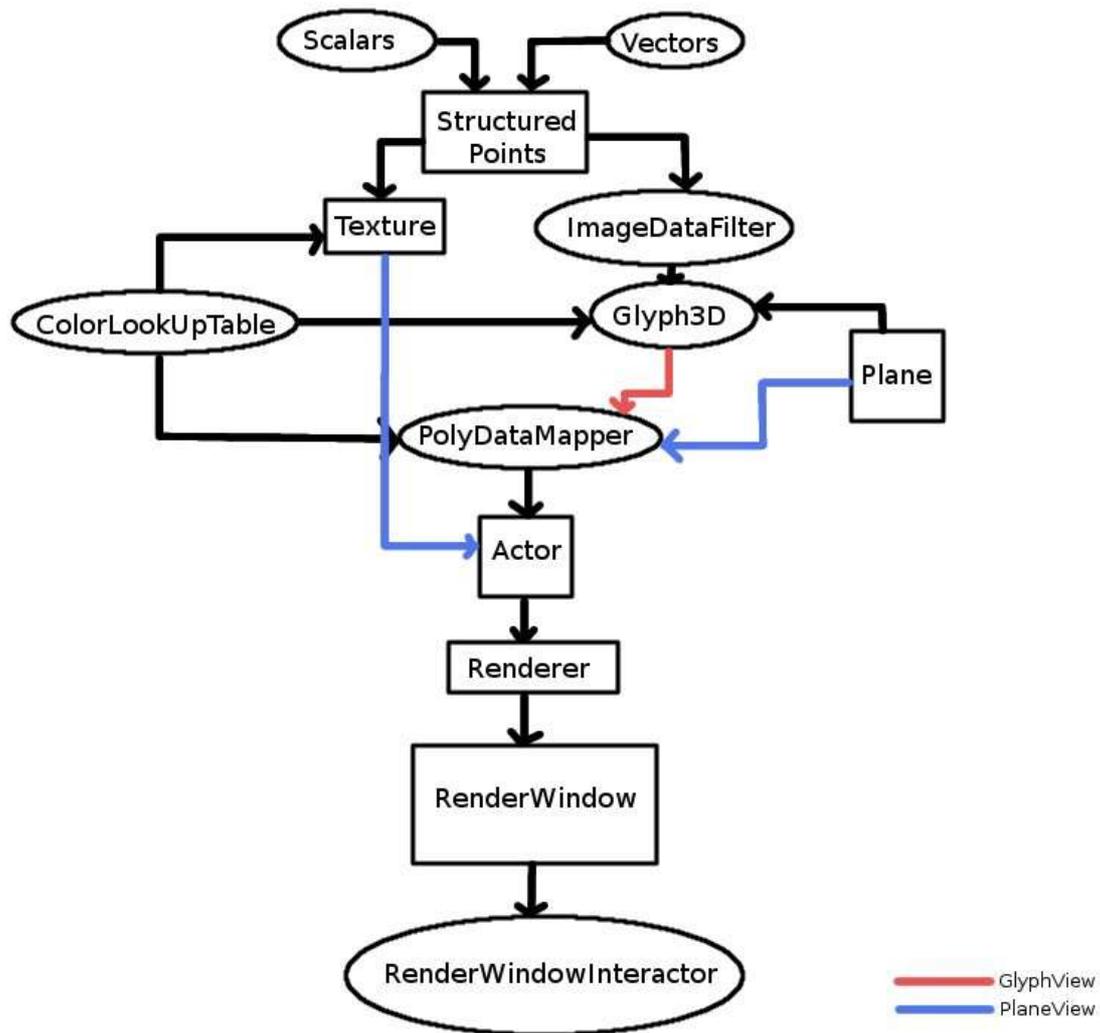


Figure 4.2: Pipeline diagram, shows the components of the application's VTK pipeline

### 4.3 User Session

Following is a description of a normal run of the application. Firstly, the user must enter the path of the XDL file and the data file. Once the XDL file is read, the hashmap of slices is created and the architecture of the FPGA is known. The activity data file then provides information about the transitions of the wires in the FPGA. This information is stored in the wire structure in the form of a list of time steps at which the wires transition. Once this data is ready, the simulation can begin. The user must then select an averaging window size and playback speed and then press Play. This will

begin the playback of the simulation and the time step variable in the VtkCanvas is updated periodically by the animation function in the SettingsDialog class.

Each time the time step gets updated, the scalar values of the structures points are re-evaluated by the *evaluate()* function. The evaluate function steps through each and every slice in the vFpga and calls the getMapValue function in vFpga. This function must then step through each wire connected to the slice (input or output). At each wire, it calculates the number of transitions by iterating through the array of time steps until the current time step or a time step greater than the current one is reached. This sum is then divided by the maximum possible transitions (averaging window - 1) to get the Transition Density ratio, which is used to determine the final metric value. The metrics of all the wires are then averaged to determine the value at that particular slice. This value is returned to the evaluation function and the scalar value at that position is set and the canvas is redrawn. *Spatial averaging* has also been implemented such that if a point is “on” (has a value greater than one) its neighbors are also turned on. Currently, a position is either one or zero to make it easier to see the movement of transitions across the chip. Once the grid is re-evaluated the colors get updated on the visualization and the process is repeated again at the next time step until playback is stopped.

Simultaneously, the GraphView also updates itself. If the current mode is transition density across the X axis, then it must iterate through all slices and do the same procedure described above independently of the VtkCanvas. However, if the current mode is net lengths then it iterates through all the nets to get their lengths. This slows down the rendering substantially and can easily be improved if the metrics are pre-calculated and stored in memory.

## Chapter 5: Experiments

### 5.1 Generating the data

One of the biggest challenges faced in this project was generating the right data. The hardware architecture is obtained from the Xilinx ISE and it is stored in a proprietary format called NCD. After much research, we were able to find the ncd2xdl tool that converts this proprietary binary format into a readable text format called XDL. A reader for this format was developed and we were able to see the architecture of the FPGA. In order to generate an xdl file from an NCD file, one must run the following command in the console (where file.ncd is the name of the ncd file):

```
xdl -ncd2xdl file.ncd
```

Next, we needed to get the activity data dumped into a text file. We found that in order to save the transition data, one must write a test bench that gives some inputs to the FPGA over some period of time (in VHDL). Once the test bench is compiled, we can invoke the ModelSim program from the command line and pipe it a text file that contained a set of instructions. A sample script could be:

```
vcd file test.vcd
vcd add *
run 100 ns
vcd checkpoint
run 100 ns
vcd checkpoint
quit -f
```

This script file will generate a VCD data file that checks the transitions of all the wires in the given simulation after two 100 ns intervals. The “vcd add \*” line adds all of the main wires of the circuit but it does not add internal wires between slices. To do this one must replace that line with the following two lines:

```
vcd add /*
vcd add /tb_sw/x0/*
```

Where “tb\_sw” is the test bench name and “x0” is the name of the main component. With this, we were able to see all the wires in the FPGA in the data file. Once we had this data, we were able to read both the XDL file and the VCD file into our application and see the results. The transitions were visible but there was no pattern. The transitions seemed almost random and it was hard to determine what was happening. Ideally, the transitions should move across the chip in a pipelined fashion. After much investigation, we determined that the rogue transitions were coming from the many internal wires and flip-flops of the circuit. The internal wires, for some unknown reason, were transitioning randomly when no inputs were passed to them. Additional changes in the C code were made to generate the design. After the changes were made, the transitions indeed moved in a pipeline along the FPGA and a pattern could be easily seen.

## ***5.2 Comparing datasets***

Once the generation of the datasets was solved, two designs of the Smith-Waterman Algorithm were implemented: (1) a design that was hand optimized by Dr. Ravindran, Dr. Mukherjee and Mr. Jong Byun to be efficient and pipelined, (2) an automatically generated design by the Xilinx ISE without any optimization. When loaded into the visualization, the two designs look very different and it is obvious which one is more optimal. On the next page is a screen shot of the optimal design followed by the non-optimal one. The red portions are active (ie. have transitions) and the green parts are inactive.

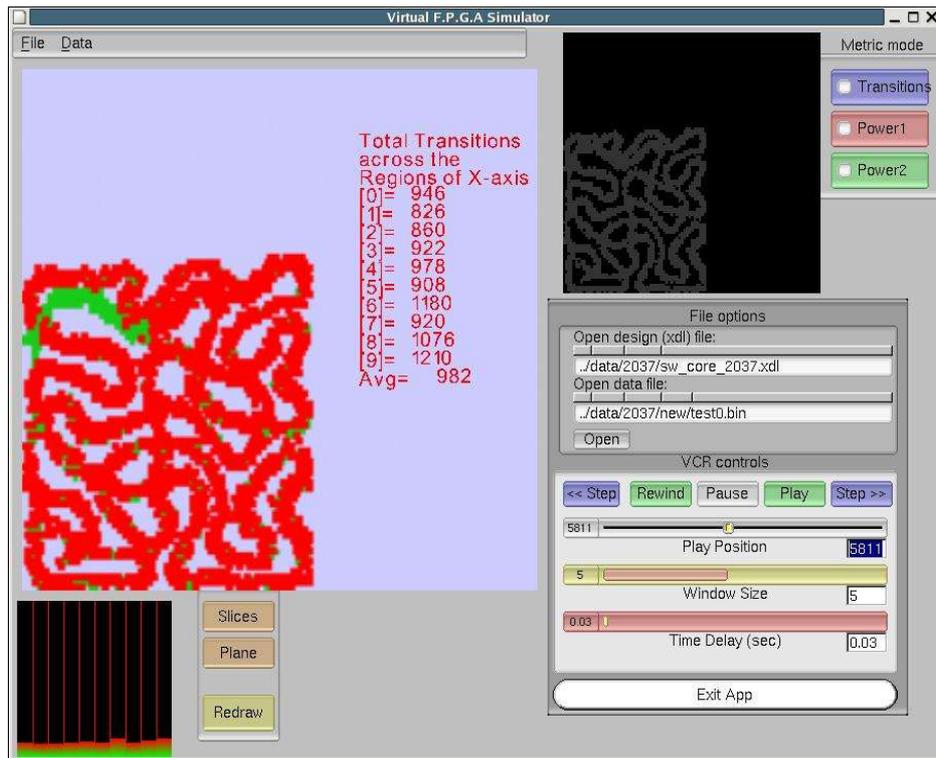


Figure 5.1: Hand-optimized dataset, 8186 slices and 8181 nets. 2037 character string

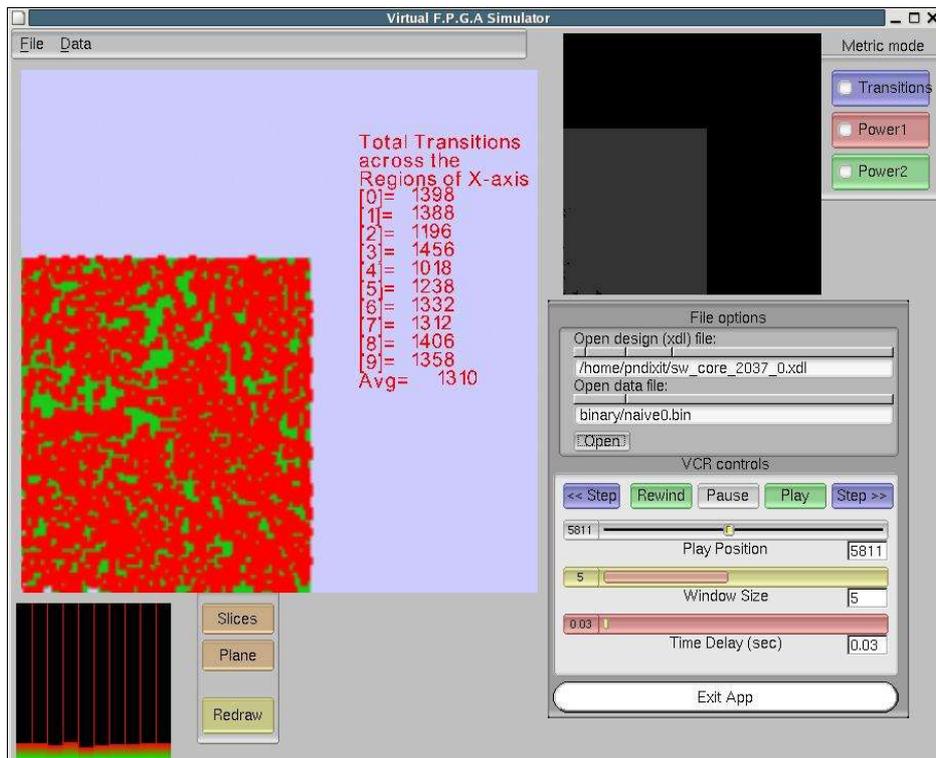


Figure 5.2: Unoptimized dataset, 14297 slices and 8181 nets. 2037 character string

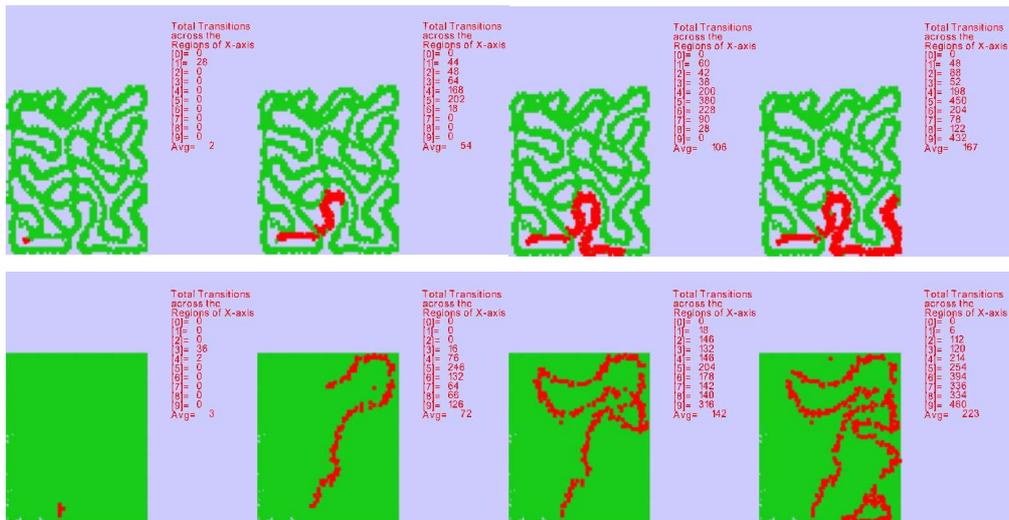


Figure 5.3: Progression of datasets, top: optimized, bottom: unoptimized

The above figure shows the progress of these two datasets, as can be seen the second dataset is not moving in a uniform manner, while the hand optimized design is very uniform and the pipeline is clearly visible. The images are roughly 300 time steps apart and from both datasets are shown in the same interval. These results were very important because they show that our visualization system is now functional.

This application remains to be tested with different algorithms and data sets. An algorithm that is not as nicely pipelined like Smith-Waterman would make the power of this tool more evident. It can be used for many different applications such as encryption algorithms.

## Chapter 6: Conclusion

Despite the many difficulties we faced in getting the data and testing the visualization, the final results are promising. The primary objective of this project was to develop the visualization system and it has been achieved. Even in its current form, the visualization tool can be extremely helpful in the mapping of an algorithm to an FPGA and in developing the hypercomputing platform. Further optimization and evaluation of the application can help make it more powerful and user friendly. The basic design flow of our computing system is in place.

There are many things that can be done to further improve the application. For example, the detailed view (GLCanvas) needs to be worked on extensively. Giving it a color map similar to the VtkCanvas can extend the detailed view. We can also make this view into another VTK view and have it display a textured plane of the selected region of interest. The graph view can also be improved by adding more metrics and making it easier to change the metric that is displayed. One of the major revisions that can be done to the system is pre-calculation of metrics. Currently, all metrics are calculated on the fly as the simulation runs, this slows down the system considerably. This also means that all of the views must calculate the metrics separately, which causes unnecessary computations. More features such as a partitioning tool can be added to the visualization system. The partitioner would enable the user to modify the hardware implementation and observe the results before actually making the change in the FPGA. The visualization can also be extended by having multiple FPGAs and adding a third dimension to the simulation.

## References

- [1] “Fast Light Toolkit (FLTK).” 12 May 2005 <<http://www.fltk.org>>
- [2] “VTK Home Page.” 12 May 2005 <<http://www.vtk.org>>
- [3] “Food Standards Agency – Genomics, Transcriptomics and Proteomics: Glossary of Terms.” Foods Standards Agency. 2002. 12 May 2005 <<http://www.food.gov.uk/science/ouradvisors/toxicity/cotmeets/49737/49750/49831>>
- [4] “Glossary of terms.” Overview of Recent Supercomputers. 2004. 12 May 2005 <<http://www.arcade-eu.info/overview/2004/glossary.html>>
- [5] “Information Visualization – Wikipedia, the free encyclopedia.” Wikipedia. 2005. 12 May 2005 <[http://en.wikipedia.org/wiki/Information\\_visualization](http://en.wikipedia.org/wiki/Information_visualization)>
- [6] “Xilinx: The Programmable Logic Company.” 15 May, 2005 <<http://www.xilinx.com>>
- [7] TimeLogic Corp., “Decypher bioinformatics acceleration solution.” 2002. 14 May 2005. <[http://www.timelogic.com/decypher\\_intro.html](http://www.timelogic.com/decypher_intro.html)>
- [8] Mathew, Binu. “1.1 The Problem” The Perception Processor. 2004. Stanford University. 14 May 2005 <<http://www.stanford.edu/~mbinu/perception/node2.html>>
- [9] “Computers and the Human Genome Project: Smith-Waterman Algorithm.” Stanford University. 14 May 2005. <[http://www.cse-stanford.edu/classes/sophomore-college/projects-00/computers-and-the-hgp/smith\\_waterman.html](http://www.cse-stanford.edu/classes/sophomore-college/projects-00/computers-and-the-hgp/smith_waterman.html)>
- [10] “vtkFIRenderWindowInteractor – cpbotha.net” cpbotha.net. 14 May 2005. <<http://cpbotha.net/vtkFIRenderWindowInteractor>>
- [11] A. Singh, L. Macchiarulo, A. Mukherjee and M. Marek-Sadowska, “A Novel High-Throughput FPGA Architecture”, Proceedings of ACM International Symposium on FPGAs, February 2000. pp.22-27.

## **Appendix**

### ***Application source code***

I have included the source code for this application in the following pages so it can be referred to for better understanding of the software.