THE SMART RECONFIGURABLE COPROCESSOR FOR FUZZY SEARCHING OF SAGE GENERATED DATASETS

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science and Engineering in the
Graduate School of The Ohio State University

By
Harrison B. Smith, B.S.E.

* * * * *

The Ohio State University
2006

Masters Examination Committee:
Dr. Stanley C. Ahalt, Advisor
Dr. Ashok Krishnamurthy
Dr. Bradley D. Clymer

Approved by

Advisor
Graduate Program in Electrical Engineering

ABSTRACT

In recent years, available genetic databases based upon the SAGE algorithm have grown rapidly in size and complexity. Along with this increased richness of data comes the need to process more data more thoroughly. In the past, such analysis was done on a relatively small scale by uniprocessor machines running generic serial code. Given the size of genetic databases today, this is no longer a viable option. In this paper, an architecture for a reconfigurable hardware based solution is presented, along with results of implementation, including performance and resource consumption. This processor is shown to be effective, efficient, flexible, and scalable. This solution will provide new processing power that will allow for more useful and detailed searches of genetics databases.

Dedicated to my parents, who paid for this document.

ACKNOWLEDGMENTS

I wish to thank my advisor, Stanley Ahalt for all his help and support in producing this document. Additionally, I would like to thank the members of my defense committee for helping me to work around their busy schedules.

I would like to thank Eric Stahlberg for all his help and insight with this project and his countless hours explaining various nuances of problem to me.

VITA

July 2, 1981 ..... Born – Naples, Florida

2004 ..... B.S.E. Electrical Engineering,
The Ohio State University

2004 – present ..... Graduate Researcher, Ohio Supercomputer
Center, Columbus Ohio

FIELDS OF STUDY

Major Field: Electrical & Computer Engineering

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Large scale genomic processing is now well into its second decade in existence[1]. In this time, several methods of gathering and digitizing genomic data have emerged as defacto standards. One of these leading methods is the Serial Analysis of Gene Expression (SAGE) method. In the last decade, SAGE has been improved, expanded, and refined into a fairly elegant and robust process. While the SAGE method for the generation of genomic data has improved greatly in the last ten years, the actual processing of the data has improved little. Largely, processing today is still done by purpose written, inefficient brute force codes, largely written in C, C++, Java or some other equivalent language.

One of the greatest improvements has been the incorporation of database technology, so that large collections of data remain orderly and searchable. However, this solves only the most minor problems relating to SAGE data processing. SAGE datasets still take a significant amount of time to process and almost all processing is done statically, with no room for SAGE processing transcription errors. The very few systems that do account for such errors are still inefficient and not very fast, or are incredibly power hungry and expensive.

While this development in bioinformatics has been ongoing, development in another area has taken great steps forward as well. In the last ten years, reconfigurable computing has grown from a laboratory novelty to a high performance solution to many problems. In the future it looks poised to grow further to become a solution to any problem.

Reconfigurable computing has its roots in the development of Field Programmable Gate Arrays (FPGAs). FPGAs were first widely available starting in 1985[2]. They began as small collections of memory that could be programmed to emulate a small logic function in a circuit. They quickly grew in size and the electronics industry started to take notice of them as a less expensive alternative to ASICs (Application Specific Integrated Circuit) in low volume productions. As FPGA costs decreased and logic capacities increase, they continue to replace more and more ASICs in system level designs. Today the computational cores of many electronics devices, such as cell phones, PDAs, and the like, are actually FPGA cores. Algorithm specific computation with FPGAs began later when the complexity and size of the chips finally reached a critical mass that allowed for sufficiently sophisticated algorithms to be implemented within them.

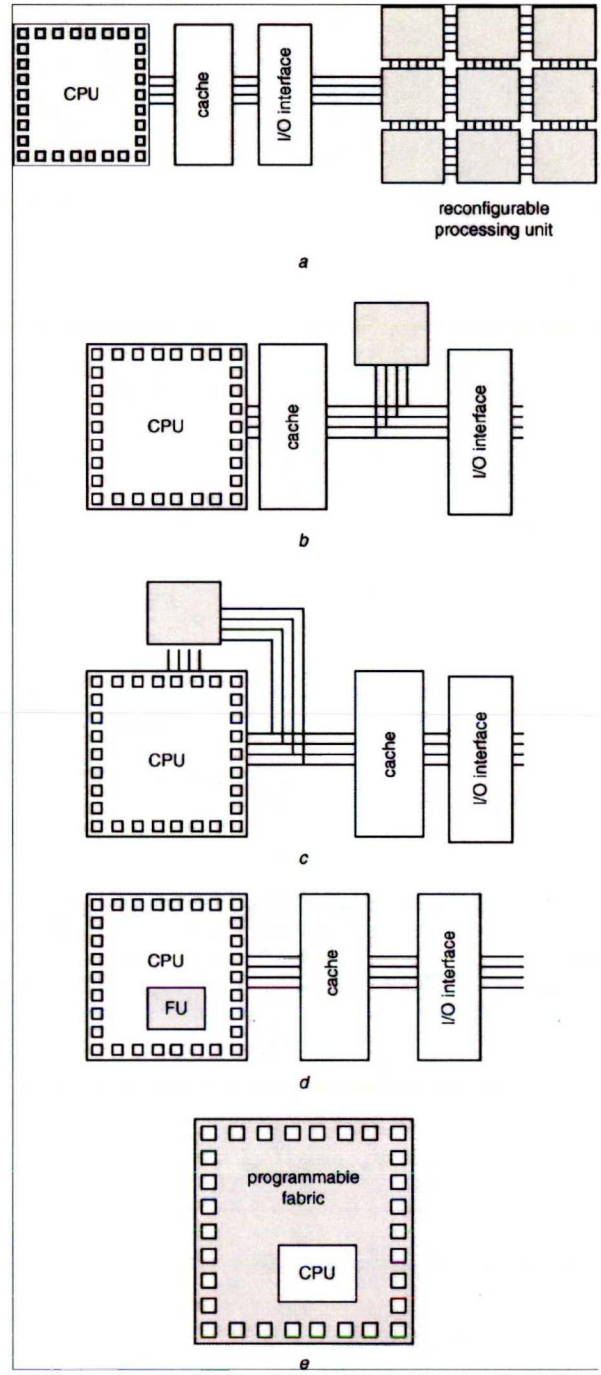

Figure 1 : Reconfigurable Computer Configurations[4]

Most recently there has been an emergence of a variety of commercial reconfigurable computing platforms. These platforms all provide reconfigurable logic via different configurations of FPGAs (see figure 1 for different classes of RC system). Access to the reconfigurable logic is provided to the user through a variety of programming environments. These systems

have been shown to be very powerful, sometimes achieving performance increases over modern CPU systems of 800x or more[3]. Currently, Cray, SGI, and many other companies are offering commercial reconfigurable computing systems. Today, reconfigurable computing is rapidly establishing itself as a valuable choice in supercomputing[4]. There are additional motivations beyond pure performance. Given that FPGA clock speeds are an order of magnitude slower than CPU clock speeds, and that more of the hardware in FPGAs is operating directly toward the computational solution, FPGA resources result in an average energy savings of 35% to 70%, depending on the application[4]. Given that modern computer rooms are limited in computation only by how much power can be brought into the building, doing more computation with less power is extremely desirable. Also, given the cost of power, a higher efficiency system results in an overall reduction to the operational cost of a supercomputing facility.

In this paper the developing fields of reconfigurable computing and SAGE data processing join to help create the SMART fuzzy searching coprocessor system. The SMART coprocessor will allow faster, more efficient processing of larger amounts data. This is done by leveraging the inhering parallelism in genetic string searching found in SAGE as well as the ability of reconfigurable computing to adapt to take full advantage of said parallelism. Hardware designs of all created hardware modules that compose the SMART coprocessor are presented. Additionally, the advantages and disadvantages of a hardware solution over a software solution are examined through comparisons of performance data, as well as additional information such as development time, portability, flexibility and so on. Current limitations of the SMART system are also presented, as well as future development paths for this project. Finally recommendations are put forth for future developers including suggestions of development behavior, pitfalls to avoid, overall design strategy where reconfigurable computing is concerned, and tools that were or would have been useful.

CHAPTER 2

SAGE BACKGROUND

Biological Background

Serial Analysis of Gene Expression, or SAGE, was conceived at the Johns Hopkins Oncology Center[5] in 1995 as a method to rapidly analyze a large number of genetic transcripts. To better understand the value of this technology, a brief overview of genetics and genomic processing is helpful.

The genome of any organism is composed of a collection of chromosomes. These chromosomes are made of long sequences of DNA, which are in turn composed of four base nucleotides: cytosine, adenine, thyamine, and guanine. These chemicals bond together to form the familiar double helix of DNA. Before any computational work can be done, actual DNA must be harvested and digitized. In genomic processing, the above chemicals are digitally represented by the letters C, A, T, and G. Series of these characters are often referred to as transcripts and individual characters are referred to as base pairs or bp. These character sequences, or transcripts, are then digitized for processing. The SAGE method is one method of gathering DNA from cells and digitizing it into character sequences. SAGE was originally developed as a way to study pancreatic cells, specifically related to cancer[5].

With SAGE, double stranded DNA within cells is first harvested and divided into smaller sections via an anchoring tag, typically with average length of 256 (or 4^4)[5]. The portion of the sequence not attached to the anchoring tag is then isolated, which provides a unique sequence associated with that particular division. This is then processed to release a small unique sequence along with the known anchoring tag. All of these small sequences are then concatenated and the full long sequence is then cloned. After this final step, the sequence is digitized (see Figure 2[6]). In this way, the presence of a small

number of a particular tag can still be captured, and, overall, the abundance of different tags, or their expression level, can be determined. At this point in the process, each tag represents a particular portion of the organisms' genome, and each tags' expression level refers to the extent to which that portion of the genome is being used. This information has lead to the classification of innumerable genes in many organisms.

The success of the SAGE method is based upon two primary principles[7]. First, that a short sequence of nucleotides is sufficient to uniquely identify a transcript. Second, the combining of many short tags into a single longer tag allows for more efficient cloning. Based upon these principals, SAGE became the first method that did not require priori knowledge of sequence significance[8]. Because of this, SAGE analysis has expanded from the study of pancreatic cancer to the study of expression levels in general. From this study, theories can be generated on the functionality of particular parts of the genetic code. Additionally, observations can be made with regard to the effects of drugs, or other stimulus, on the cell[9].

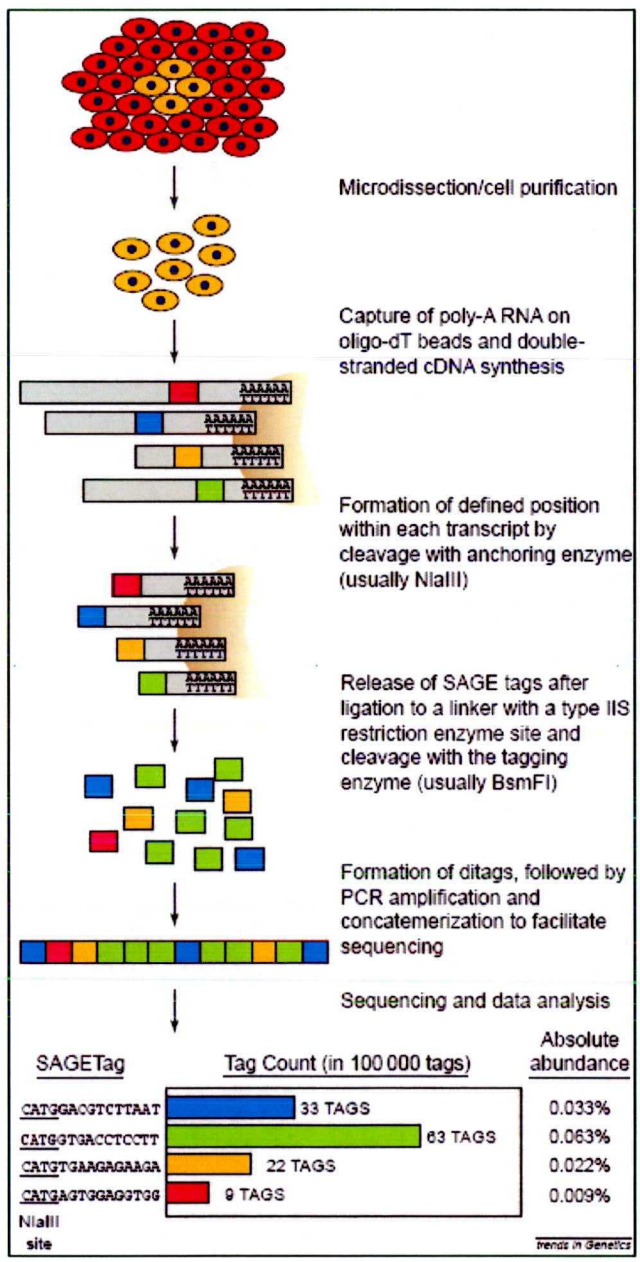

Figure 2 : The SAGE Process[6]

Originally, SAGE used sequences of 9 to 10 nucleotides to distinguish transcripts; more recently, sequence lengths have been increased to as many as 21 nucleotides through the Long Sage[9] and RL-SAGE[10] processes. It is foreseeable that

sequence lengths could increase in the future to 25 or perhaps even 30 base pairs. Far beyond these lengths, very little further distinction can be gained as the number of sequence combinations become so large that they begin to loose meaning. In addition to increased tag length, the process as a whole has been improved. Today, genetic information can be derived from amounts of genetic material nearly one hundredth the size of those in the original SAGE process[10]. Additionally, the process as a whole has been made more robust and less prone to errors[10].

As the SAGE process has grown more robust, well developed, and accepted, the amount of SAGE data available has grown accordingly. Fortunately, since its inception, SAGE data has been stored in databases, or at least database-like libraries. Thus, the data is well organized and widely available. There are now a great many examples of SAGE data repositories including the Gene Expression Omnibus[11], the Saccharomyces Genome Database[12], GeneBank[13], and many others.

Computational Background

Much of the history of SAGE computational analysis is severely lacking in technical sophistication. This may be primarily due to the background of SAGE computational methods largely being focused upon the statistical analysis of generated data sets, rather than the computational methods used to create and search these datasets. In searching for research relating to the computational analysis of SAGE data, many papers were discovered such as [14], which discusses possible biases within SAGE data sets. This sort of statistical computational analysis is outside the scope of this work, but has seemingly been the focus of much of the effort regarding computational SAGE techniques.

In many cases such as [15], papers reference [5] with regard to their computational approaches. Beyond that, analysis is done primarily with nondescript purpose written code. For instance, in [16], "A series of JAVA and C programs were designed for [the SAGE] analysis" is as in-depth a description as the researcher provides. Likewise, in [17], the author states only that, "Computational programs were designed using java language for the [processing]." Again in [18], the author merely covers that Perl was chosen for its "versatile portability" and says nothing of how the algorithm was

actually implemented. In all of these cases, the vagueness of description leaves one to conclude that these programs are most likely simple, unoptimized, serial applications.

More recently, as the issue of volume processing of SAGE data has become more pressing, more attention has been paid to this topic. Among the efforts to enhance the performance of analysis is the Gene Identification and Sequence Topography suite (GIST) produced by the University of Chicago[19]. This package performs exact matching between genetic data in parallel. Additionally, a suite of tools known as Bioconductor[20] has been produced by a collaboration of many universities. This software package provides extensive functionality beyond the processing of SAGE data, but it seeks to provide a common platform of bioinformatics based computation.

Finally, several hardware designs for biosequence processing have been developed. Quite a few special purpose architectures using sequences or identical processing elements, known as systolic arrays have been developed[21]. However, success has been limited due to lack of flexibility of these pure-hardware devices. More successful is the FPGA based SPLAH2 system. However, the SPLASH2 system was not built explicitly for the processing of biosequences, but rather for the implementation of systolic array systems[22]. Additionally, the core designs of the system are not open for development and the FPGA hardware of the boards themselves are becoming grossly outdated. In addition to the SPLASH and SPLASH2 systems, the Decipher system from TimeLogic is another option, though is suffers many of the same problems as the SPLASH systems.

All the above solutions are still not fully sufficient for two primary reasons. First, the SAGE process is not perfect. SAGE, being a biological and thus analogue process, is subject to noise and therefore inaccuracies. An acceptable error rate for usable data is often considered to be around 1 in 100[18]. In the event of such an error, it is desirable to still be able to return a result, noting the nearness of the match. Given the cost of the creation of SAGE data[10], allowing for slight errors in processing allows for the usage of greater amounts of data, making money spent on generating said data more effective. Today, only a few software solutions that allow fuzzy matching exist, mostly utilizing the Smith-Waterman algorithm[24] or the BLAST algorithm[25]. Most notable among these is the SAGESPY application developed at OSC for the Cray X1 system[26]. This

application uses a BLAST search approach to SAGE data processing and incorporates the ability to perform fuzzy matching. While SAGESPYs performance characteristics, even during fuzzy matching, are quite promising, the execution of the application depends heavily upon the special hardware available only on the Cray X1, which is quite an expensive computer system. Performance of SAGESPY on other platforms is considerably slower.

Secondly, and more critically, as sequence lengths increase and databases grow, computational demands will certainly increase. Scientists will want to perform more detailed searches for more sequences in larger databases. For software solutions, updating to changing input parameters will not be a problem. However, computation times will increase riotously. Conversely, for hardware solutions available today, the user will either be forced to upgrade to new hardware, in the case of traditional hardware, or will be required to wait for an upgrade to their FPGA based solution, as the current FPGA solutions do not have open cores, if indeed an upgrade is even possible. Obviously, none of these situations are desirable. Combining all these considerations, even the relatively short search times involved in most work today will increase to unreasonable lengths.

The SMART coprocessor is a more cost effective and efficient solution than SAGESPY, while being nearly as powerful. Additionally, SMART is a more modern and focused solution to genomic data processing than the hardware solutions existing today. Further, as the core is open, it is user maintainable and more highly adaptive to changes in the SAGE process. Finally, SMART was designed to be as scalable as possible to allow for expansion of the system as new technology arrives and further functionality is needed.

CHAPTER 3

SMART PROCESSOR DEVELOPMENT

Problem Requirements and Conditions

To best understand the decisions made during the development of the SMART coprocessor, understanding the design constraints of the problem itself is necessary. These constraints consist of the format of input data, the information needed from the processor, and the relative likelihood of important processing events. An examination of these constraints is presented here before going into greater depth with the SMART design.

First, consideration must be given to the system the application will be implemented on. The Cray XD1 is a cluster-type supercomputer system with optional reconfigurable hardware. Computational nodes consist of dual Opteron processors connected via a HyperTransport interconnect. This interconnect also services the reconfigurable hardware of the XD1. On the CPU side of development, a software interface is provided by Cray which implements basic functionality. It allows for all of the memory within the FPGA to be mapped into CPU memory space and, also, for up to 1GB of CPU memory to be made available to the FPGA via a dynamic memory allocation (DMA) region, which allows for high speed burst transfers between the FGPA and CPU. Memory attached and within the FPGA is fairly limited, with 16MB available in static RAM attached to the FPGA and additional smaller block rams integrated into the FPGA itself. Memory attached to the CPU is much larger, ranging from 1GB to 8GB. Cray provides several hardware cores for the developer. These cores provide access to a 64-bit wide data bus to CPU. The FPGA itself is capable of running at up to 199MHz.

Next consider the data format that the SMART coprocessor will be required to take as an input. The SAGE process produces a tremendous amount of textual data that

represents sequences of nucleotides. This data is almost universally stored in an ASCII file format specifically created for protein sequences known as the FASTA format. In general, query strings (Q's) are relatively short, from 9 to 21 characters[5],[9], which are also referred to as base pairs or bps. In a typical search, a user might wish to search for anywhere from hundreds to thousands of Q's. These Q's will be compared to all possible substrings of a very large target sequence (or T). Target sequences are typically between tens of thousands and hundreds of thousands, but can be as short as a few thousand or as long as a millions or more. A typical search will only involve a few, possibly only one, T string. Both Q's and T's are stored in ASCII FASTA files.

Another consideration is output data from the processor. The SMART coprocessor will be looking for exact, as well as near, or fuzzy, matches with an error range of 2 characters between Q's and a T substring. For any given match, it is important for the user to know which Q matched a substring of T, the position of the substring, and the nearness of the match. It should be noted that any given search can be broken down into a series of smaller searches by either dividing sets of Q's into smaller subsets or dividing T strings into substrings. Likewise, many smaller searches can often be combined into a single larger search in the opposite way.

Finally, briefly consider the statistics involved in genomics searching. FASTA files encode strings of nucleotides by representing each of the four chemicals they are composed of by one of four letters: A, C, T, or G. Based on the assumption that these strings are purely random, the probability of finding an exact match between a Q of length n and a T of length m is upper bounded by:

$$E(number\ of\ exact\ match\ in\ T) \leq \frac{m}{4^n n}$$

For n = 9, and m = 10,000,000 this evaluates to approximately 10. Thus, for the shortest meaningful Q, and a relatively long T, the approximate expected number of exact matches is 10. During SAGE processing, transcription errors can occur and therefore fuzzy matching that would allow for no more than 2 transcription errors per match was deemed an important objective for the SMART coprocessor. Clearly for fuzzy matching the probability of a match will increase. In fact, it is upper bounded by:

$$E(number\ of\ fuzzy\ matches\ in\ T) \leq \frac{1+3n+3^2\sum_{i=1}^{n}(n-i)}{4^n} \bullet \frac{m}{n}$$

For the same values of n and m as above, this evaluates to approximately 1492 fuzzy matches, which means one result will be found for approximately every 6700 characters of T that are processed. However, it should be noted that genomic data is not purely random. In fact that a SAGE experiment is being performed is, by itself, enough to increase the likelihood of matches significantly. However, in practice, this increase is no more than a single order of magnitude.

Design Considerations, Goals, and Decisions

The above information leads to a collection of design considerations, goals and decisions. First among these was to keep all input and output data in processor memory space. While the memory available within the FPGA would most likely be sufficient to store all the needed data, the overhead involved in accessing this memory from the FPGA, as well as the cost in time of writing the data from the processor memory to the FPGA memory, was too great. More pointedly, data will not be permutated upon in this application and, therefore, each word of memory will only be accessed once. As such, moving it from CPU memory to FPGA memory is superfluous when one could move data from CPU memory directly into the computation path of the FPGA at the same cost.

The next consideration has to do with the format of input data. Given that, in the XD1 architecture, the bus between the FPGA and the CPU is only 64-bits wide and ASCII characters are 8 bits each, this allows only eight characters of data to enter the coprocessor per clock cycle. In the early stages of development, it was desirable to reduce the likelihood of the coprocessor being starved for data. To this end, it was decided that the coprocessor should take as input compressed FASTA data. So long as the compressed characters are two bits apiece, the translation from ASCII to binary is irrelevant, so long as it is globally consistent. In this implementation, a FASTA compression algorithm from the Cray BioLibrary[27] was used. Still, given the sensitivity of this application to data starvation, it was decided that the coprocessor should still be able to deal with data starvation periods of arbitrary length. Additionally, compressing the FASTA characters to 2-bits each will help reduce the size of our comparator, as the

comparison of characters will only deal with 2 bits rather than 8. However, the inevitable trade off is additional processing time on the CPU prior to FPGA execution. In the future, FASTA file compression may be moved into the SMART coprocessor to further reduce CPU processing burden. This addition to the coprocessor could be accomplished with almost no discernable effect on performance of the coprocessor.

Another consideration with input is the length of Qs. Given that different forms of SAGE produce different length Qs and, more importantly, that the length of these Qs is generally increasing, it was decided that the coprocessor should be able to accommodate Qs of length greater than the largest generated today. To date, some of the most advanced SAGE processes are producing Qs of length 21bp[9].

Additionally, given that the Qs are very manageable in size (from 18 bits to 48 bits to date) and in quantity, and T strings tend to be unwieldy large, it was decided that the coprocessor should hold a number of Q values and have a single T string streamed through it. Because of this, it was decided that the coprocessor should consist of a large number of identical processing elements (or PEs) each of which would hold a single Q value, and a single controller to orchestrate their configuration and the movement of data to and from the CPU. Then, all the T characters would be streamed through these elements, with each element performing comparisons in parallel. By using an array of identical PEs, design and optimization efforts were focused on one unit, which will then be replicated, greatly increasing the effectiveness of time spent improving the PEs. This overall design pattern was influenced by both systolic array systems, as well as dataflow architectures.

When considering the required output, note that the Q value and the T substring are not needed. This allows the processing engine within each processing element (or PE) to consume these values, passing on only how close of a match the particular T substring is to the Q value. However, each PE must be aware of the position in T from which the substring was taken. Additionally, each PE must, in some way, make its matches distinct. To accomplish this, each input Q value is assigned an ID based upon its position in CPU memory. When these Q values are loaded into the PEs, the corresponding ID is loaded into the PE as well. All results generated by the PEs are

tagged with their IDs, so that the Q value can be retroactively matched to results by the CPU.

Finally, consider buffering of the data generated by each PE. As each PE works in parallel with all the others, it is possible for results to be generated simultaneously in all PEs in the coprocessor. Therefore, it will be necessary to buffer results in the PEs, and create a reporting system that will ensure that each PE can report results, without worrying about bus contention and without worrying about starving any PE of access to the bus. When considering the likelihood of a match in a given PE, it becomes clear that buffers internal to the PEs would not need to be very large, even for very small Q strings and extremely large search strings. In the extremely unlikely situation of results generated by a PE overflowing the buffer, it will be sufficient to signal this condition, as finding such a high concentration of a particular string is, more likely than not, going to either negate the entire experiment or, alternatively, make the actual density meaningless in comparison to other statistics.

Finally, consider that the SMART coprocessor is using a serial chain of PEs and that scalability is of critical importance since as FPGA logic capacity grows, increasing the PE chain length will be highly desirable. Creating a linear bus by sending results back down the PE chain to the controller seems the most obvious solution; however results could be generated in such a way that there would be contention between the PE and its neighbors further down the chain. Creating logic to handle this contention in every PE is likely to be costly. Further, position of the PE in the chain would affect the desired result of the arbitration logic, making the PEs non-uniform. At the same time, a shared bus system would not work either, because any arbitration logic would grow with the number of PEs in the system, disrupting the scalability of the system as a whole. In the end, the system uses a shared bus, but access to the bus is granted by a token which is passed along a token ring between the PEs. In this way the system scales nicely and a minimum of logic is added to every PE.

Software Description

Software Overview

The SMART software package will be as streamlined and as simple as possible. Ideally, the user will only need to supply FASTA input files, one containing all the Qs they wish to search and all the T strings they wish to search for them in. At present the software is only implemented to accept up to 10 Qs, and a single T string. This present limitation is purely artificial and will exist only as long as hardware debugging is taking place. Once debugging is completed, it should be possible to have the software initialize multiple runs of the coprocessor in the most efficient way with regard to memory and disk access.

Overall, the software application is quite small and most of the complex programming has been taken care of via software libraries available as part of the C standard or provided by Cray specifically for accessing the FPGAs on the XD1. The libraries used in the software are detailed in the table below:

Library Name	Functionality
argp.h	Command line argument parsing library, used to easily deal with command line input
fcntl.h	File control options library, used to make code more readable and maintainable
stdio.h	Standard C I/O library, used for accepting user input and providing command line feedback.
stdlib.h	ISO standard C library, used for memory allocation algorithms
ufplib.h	CRAY provided API for the accelerator FPGA, used to control, configure, and run the FPGA coprocessor
unistd.h	Standard Symbolic Constants and Types

Table 1: Software Library Descriptions

Software Details

The software operates in the manner described below. The software first initializes the FPGA, and loads the hardware core into the FPGA. Then, the software creates a shared memory space between the FPGA and CPU and loads the Q and T FASTA files into this space. These FASTA strings are then compressed via the Cray BioLibrary. The software then signals the coprocessor to begin execution and polls the

coprocessor for a completion signal. Please refer to the Software/Hardware interaction diagram for a more complete picture of computation flow.

There are two likely changes in a future version of the software. First, compression from FASTA into 2-bit binary representation will likely be moved to the FPGA, assuming that the bandwidth between the processor and the FPGA is sufficient. This was not done initially due to fears that bandwidth between the CPU and FPGA might not be sustainable. Compression reduces the bandwidth needs of the coprocessor by a factor of 4. It would be beneficial to move compression to the CPU as it would take up very little space and not add noticeably to computation time. Also, the FPGA is capable of raising an interrupt for the processor and this could be used to signal completion rather than busy waiting. This has the obvious advantage of allowing the processor to perform other actions while waiting on the coprocessor.

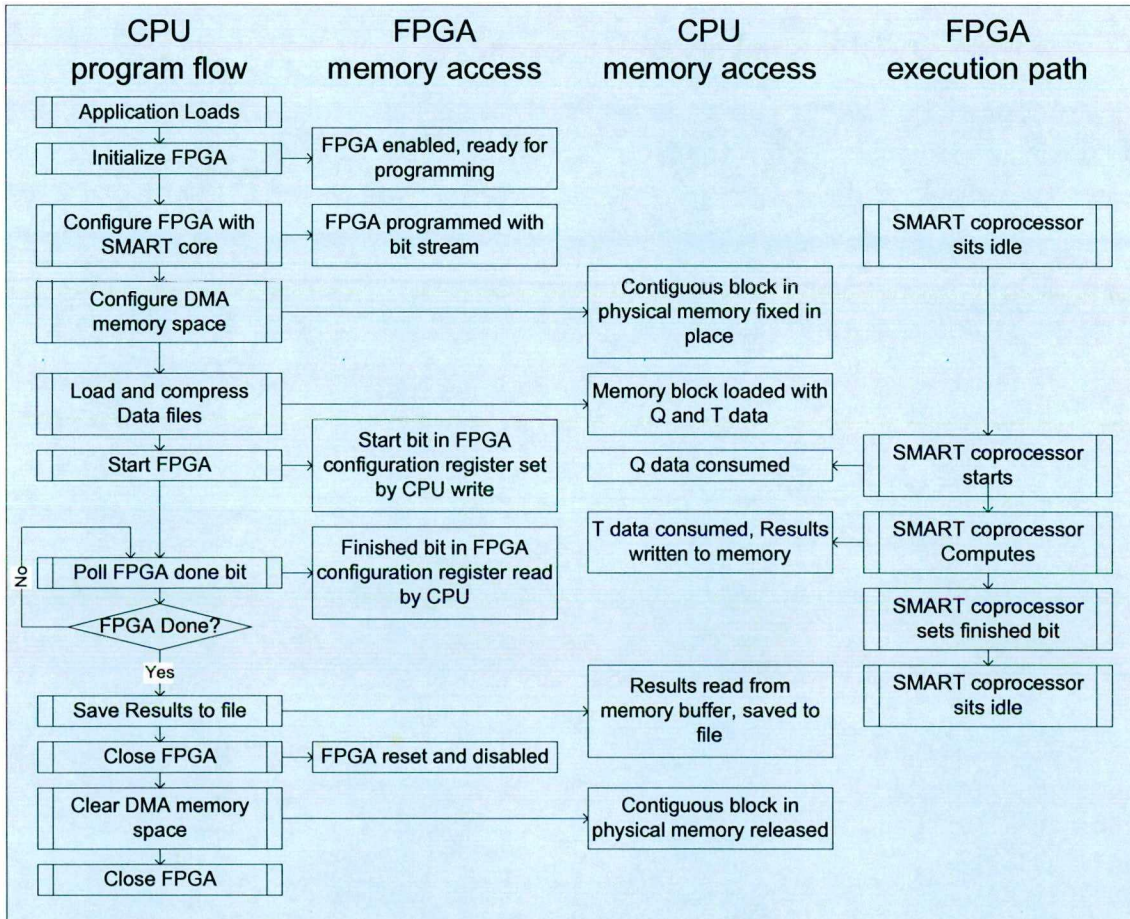

Figure 3: Software Hardware Interaction Diagram

Hardware Description

Full System Overview

The high level structure of the acceleration system is that of a serial string of custom designed processors controlled by a single controller tied into the Cray provided hardware. The coprocessor can accommodate multiple Q's, dependent upon the size of the FPGA provided with the XD1. As the XD1 at OSC Springfield is equipped, each reconfigurable processor can theoretically search a T of one hundred and eighty million bps per second for sixty Q's. Details of both the processing elements and the controller are below.

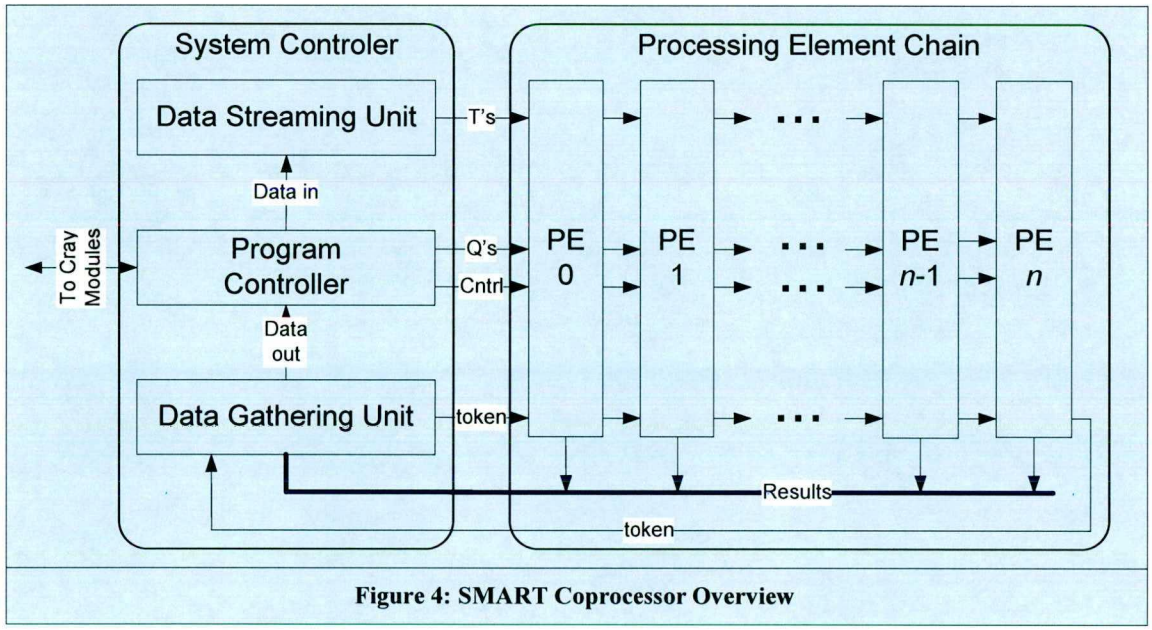

Figure 4: SMART Coprocessor Overview

Processing Element

The workhorse of the accelerator is the long chain of processing elements that will take up the majority of FPGAs real estate. As a linear chain, most of the inputs and outputs of the unit are dedicated to the passing of data through the unit. There are a few outputs of each PE that are attached to a shared bus and are used to send computational results back to the system controller. The PE requires some control signals, which originate from the system controller and are passed along the PE chain. Additionally, the PE requires a Q value and an ID value. Finally, the PE requires one character of the T

string per clock cycle in order to perform its task of fuzzy string comparison. All of these needs are handled by the Dataflow subcomponent of the PE.

Once all of this data is available to the PE, it is capable of performing one fuzzy comparison per clock cycle. This computation takes place in the Popcount subcomponent of the PE. Each clock cycle, the Popcount unit receives the Q value of the PE and the current T substring present in the PE. These values are fed into a pipelined unit that computes the number of mismatches between the Q and T string. The pipelined Popcount unit outputs the number of mismatches six clock cycles later. Additionally, it delays a number of control signals so that the Saver subcomponent knows the state the machine was in when the T string was first latched into the Popcount unit.

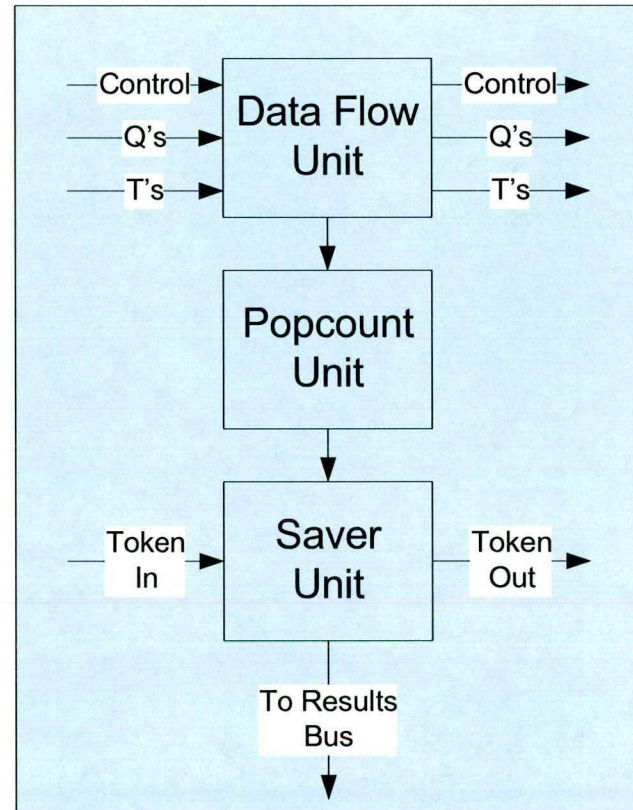

Figure 5: Processing Element

The signals from the Popcount unit are then analyzed by the final unit in the PE, the Saver subcomponent. This unit contains a counter that keeps track of the current position in T. It saves the position and match quality of results computed by the PE. Additionally, whenever the Saver unit receives the token, it may report a single buffered result back to the system controller. Below are more detailed descriptions of each of these three units.

Processing Element: Dataflow Unit

By dividing the PE into three subcomponents specific functionality is isolated. Within the Dataflow unit all the buffers and logic needed to control the passage of signals down the PE chain, as well as logic involved in the initial configuration of the PEs with Q

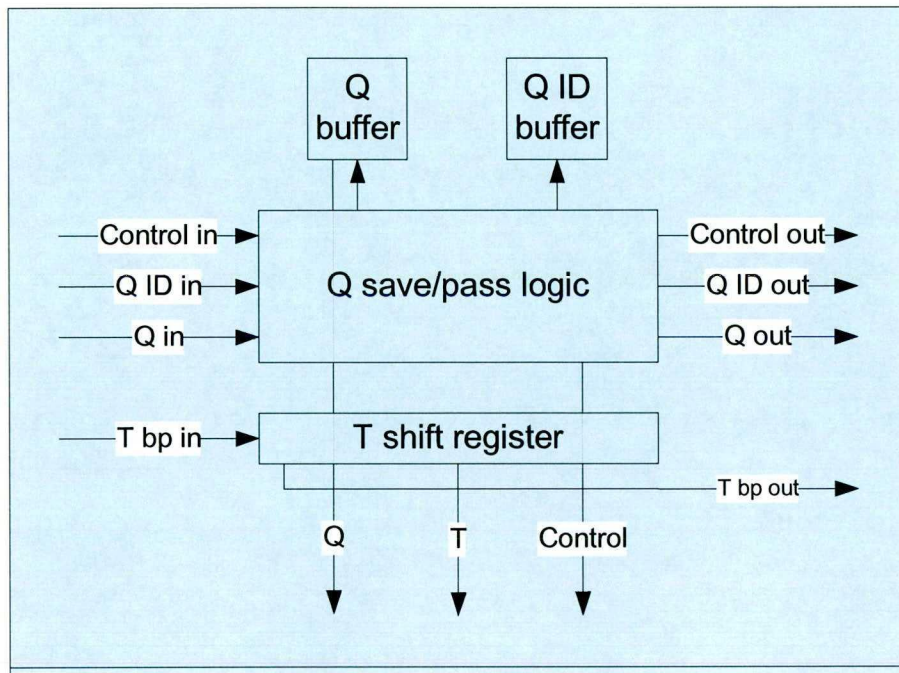

Figure 6: Data Flow Unit

values and ID, is located. The unit is mostly composed of buffers, including 64 bits to store a single Q, 64 bits to hold 32 characters of T, and a variety of other buffers for control signals. Early in the design of this unit, a 64-bit wide bus was added specifically for the passing of Q values down the chain. This not only reduced the latency involved in loading Qs, but also greatly simplified the logic needed to do so in the PEs.

All communication between PEs takes place in this unit (with the small exception of the token), and thus, through the careful design of this unit, it is ensured that the PE chain will be highly scalable. Also, due to the design of the Dataflow Unit (and the results reporting system), the system can be expanded simply by adding additional PEs to the end of the PE chain. In fact, the only current limitations are the size of the FPGA itself, and an arbitrary limitation of the ten bit ID value. The ID value could easily be made much larger, allowing for thousands of PEs.

Often, serial chains such as this result in higher latency since data has a correspondingly longer path to travel. In our case, the increase in latency is insignificant compared to the execution time of the application overall. The latency of the chain overall is a linearly dependent on the length of the chain. The design has been optimized so as to minimize the overall latency by not streaming the T values all the way through the T buffer in each unit, but rather have T characters back out again immediately as well as down the buffer (see figure 5). Moreover, the added latency is going to be measured in hundreds of clock cycles, while execution time will be measured in hundreds of thousands of cycles, dependent on the length of T. An additional advantage of streaming the T values in this way is that all values, data and control, that come into the PE on clock *x* leave the unit on clock *x+1*. The only exception to this is when the PEs are being configured with Q values. During this initialization step, PEs only pass along Q values after they themselves have received and stored a Q value. This greatly simplifies the decision logic within the PEs that involves their initialization.

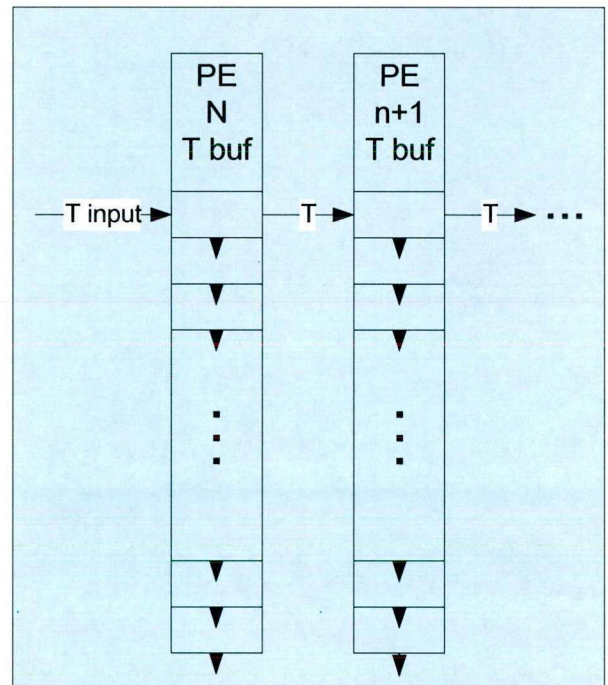

Figure 7: T Buffer Shift Register

Processing Element: Popcount Unit

Within the Popcount unit are all the computational components of the PE. The Popcount pipeline is six stages deep. The first stage compares the individual characters between Q and T and generates a 32 bit vector, wherein all mismatches of characters between Q and T are represented by 1s and all matches by 0s. The subsequent 5 stages of the pipeline sum all the bits of this vector together, generating a total number of ones, or

the "population count". This unit was fully pipelined in order to allow it to achieve a maximum clock rate of 200MHz (the highest supported clock on the FPGAs available in the Springfield XD1). In that this unit is fully pipelined, at any given moment it is performing six comparisons in parallel, so, in effect, it still performs one comparison per clock. Also, there is additional latency created by pipelining these units since a single comparison takes 6 clock cycles. However, the six extra clock cycles are again inconsequential when compared to the hundreds of thousands the full processing will require.

In a future version a 7th stage will be added to the pipeline that will allow for the masking of bits. This will allow the end user to use Qs of length less than 32bp. Also, experimentation has shown that by collapsing certain stages of the pipeline together, resource usage can be reduce without lowering the clock rate below the 200MHz barrier that the FPGA sets forth. Since the core computational workings of the PE are all contained within this unit, it would be possible to swap out this unit with another that implemented another search function. In this way, compartmentalizing our design has added some degree of reusability to it.

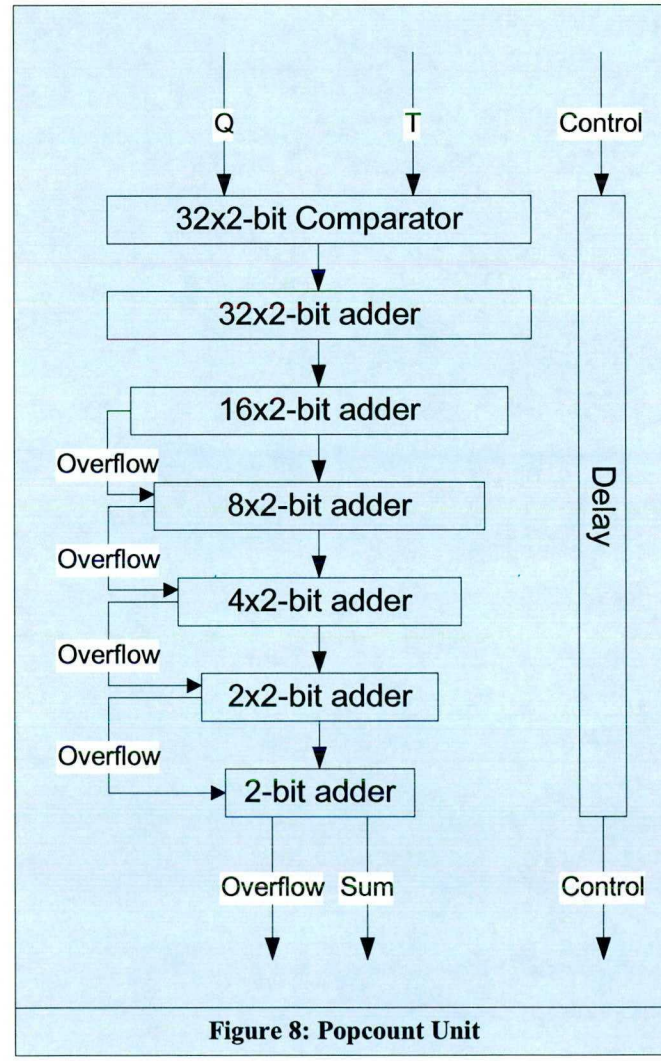

Figure 8: Popcount Unit

Also within the Popcount unit are a series of buffers that delay the control signals such that the subsequent Saver unit will have the state of the machine corresponding to the current output of the Popcount unit.

Processing Element: Saver Unit

The final component of the PE is the Saver unit. Within this unit is all the logic that analyzes the output of the Popcount unit. Additionally, it contains logic that keeps track of the current position of the T substring as it is streamed through the coprocessor. Using these two logic components, it decides every clock cycle whether or not to buffer the current position of the T substring as a match to the Q value within the PE or to rule the result a mismatch. The buffers within the PE are implemented using the numerous block rams within FPGA and can contain up to 128 results at once. The buffers are also circular, that is, data is written to locations sequentially, and when the last position in the buffer is reached, the process beings again at the first postion. This eliminated the need for complex logic to keep track of empty and full locations. Each PEs saver unit uses one block ram as a buffer and, as the FPGA has more block rams than its capacity for PEs, this is clearly not an issue with scalability. The buffers can be so small due to small probability of a match occurring. The probability of a match is certainly higher than pure statistics suggests, since it is conditioned on the fact that meaningful genetics data is being searched. However, it is still very small, and software solutions have shown that well less than 0.1% of T substrings result in

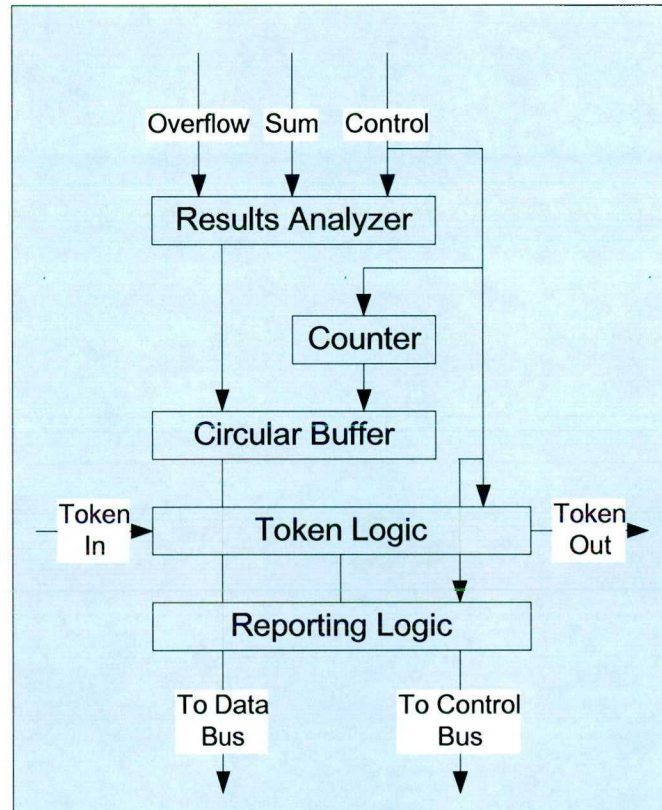

Figure 9: Saver Unit

matches[26]. Given the statistical unlikelihood of a buffer overflow, it was decided that any run in which one might occur would have very little meaning anyhow, and simple writing a special value to the buffer in these cases to signal to the user that overflow occurred would be a sufficient response. In this way, the user can make adjustments as needed for future runs.

Also, given the unlikelihood of a large number of matches, a single bus was used to connect the controller to all the PEs. Control of the bus is granted by a token originated at the system controller and passed between the PEs. Whenever a PE receives a token, it may report a single value back to the system controller before passing on the token. In this way the PEs dependence on their buffers are further reduced and at the same time, a bus arbitration system is provided that ensures there will be no starvation while requiring a minimum of logic to be present in the PEs.

System Controller

The most complex component of the fuzzy matching system is the main system controller. This controller interfaces with both the PEs and several Cray provided modules. To the PEs, it is responsible for providing configuration and, once the computation has begun, a constant stream of input data as well as collecting results during the course of execution. When interfacing with the hardware modules provided by Cray, the controller is responsible for responding appropriately to initialization and configuration data, requesting reads and writes to and from CPU memory space, and alerting the CPU when the computation is completed. The controller interfaces with several Cray modules to achieve these objectives (see table 2 below). The system controller is composed of three sub modules each of which contain the hardware related

Core Name	Purpose	MHz	Slices
dma_if	Provides DMA based access to CPU memory	189	727
reg_if	This register unit was modified to become the register controller discussed below	818	50
rt_client	This unit provides routing to different CPU addressable regions of the FPGA	270	468
rt_core	This unit abstracts the rapid transport interconnect	199+	n/a

Table 2: Cray IP Modules

to a specific function of the controller. These three subcomponents are the Data Collector, the Data Streamer, and the Main Controller. These three components will be discussed in detail later.

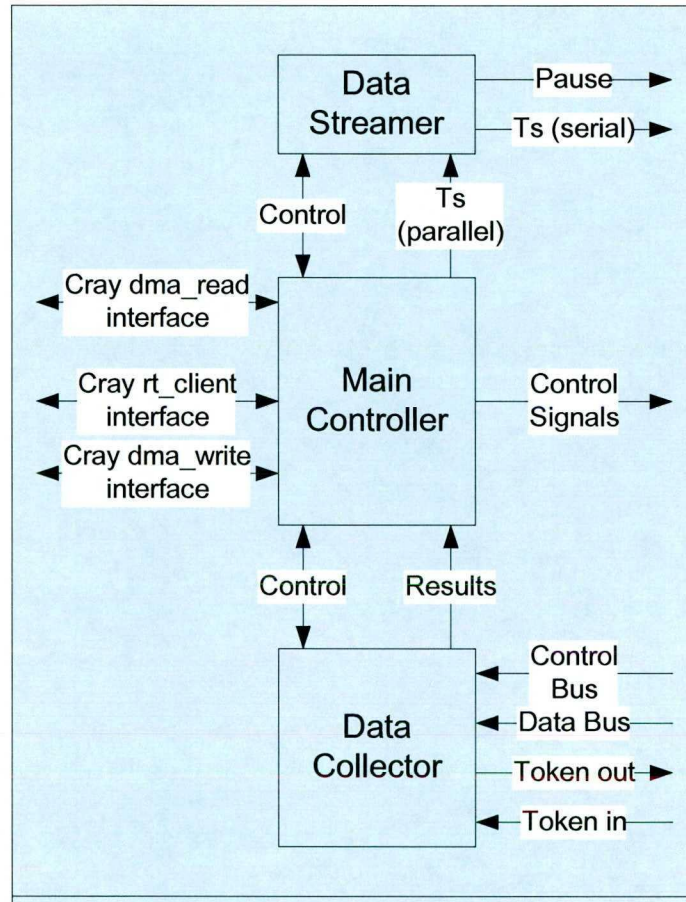

Figure 10: System Controller

In designing the system controller, Cray modules were critical. Making use of these Cray provided modules helped ensured proper functionality. What's more, these components implemented some of the more difficult hardware in the design in an efficient and general way. They are highly reliable, easy to use, and do not present a significant bottleneck in the system. All hardware communication interfaces were abstracted by Cray modules to provide a more clear and intuitive interface. Additionally, the routing of data transfers to and from the CPU and from within FPGA memory units were handled by Cray modules.

By dividing the system controller into three subcomponents, development and testing were not only simplified, but the reusability of the design was increased. The Data Streamer and Data Collector units only interface with the Main Controller. All connections between Cray IP cores and the coprocessor are contained within the Main Controller. In this way, moving the system to another platform should only involve the modification of this subcomponent.

System Controller: Data Streamer

The Data Streamer unit is an intelligent parallel to serial converter that keeps the PEs supplied with elements of the target sequence at a rate of one character per clock cycle. It relies on the Main Controller for parallel T data retrieved from the CPUs memory space. Its intelligence comes from its awareness of the data contained within its self. Rather than simply shifting whatever is contained within it, the Data Streamer tracks where valid and invalid data is within its 32 bp buffer. When less than half the buffer is valid, the Data Streamer starts requesting additional data from the Main Controller. Should the buffer ever empty fully, the Data Streamer asserts the global pause signal, sending the PEs into a suspended state, awaiting new data.

The Data Streamer was created to be as small as possible while still providing a limited awareness of its own state. The simplest way to achieve this was with a 64-bit (32 bp) buffer and a small counter. As the buffer is shifted out to the PEs one bp at a time, the counter counts down. When new data arrives, the counter is reset. Logic signals for new data when the counter drops below the halfway mark. This ensures that the main controller has sufficient time to retrieve and route data to the Data Streamer. Also, a second set of logic pauses the system whenever the counter is less than 1. The actual logic that determines when to send data to the Data Streamer is all contained within the Main Controller.

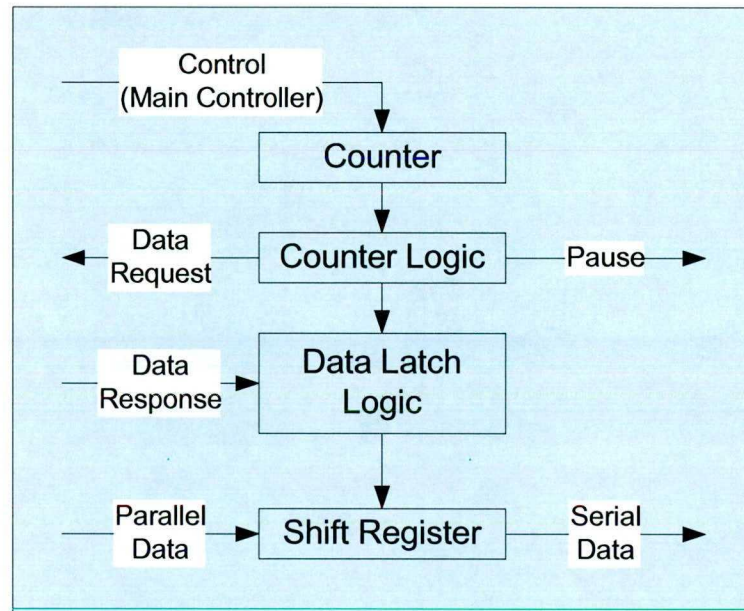

Figure 11: Data Streamer

System Controller: Data Collector

The Data Collector is a simple token generator and a circular data buffer system. It creates the initial token that is passed between the PEs. It also receives the token from

the final PE in the chain and passes it back to the first PE again. It monitors the shared reporting bus and control lines that are connected to each of the PEs. The PEs signal as they receive the token whether or not they have any results to report and, if they do, they latch this data onto the bus for two clock cycles. In this time, the saver unit buffers the data into the next available location in its circular buffer. This buffer is again smaller than one might imagine, due to the small probably of many matches occurring at once, but also because the Data Collector can request one write per clock cycle back to the CPU via the Main Controller. Therefore, baring bus contention between the CPU and FPGA, the Data Collector can only receive one new result every two clock cycles, but can report 2. So buffering is solely to compensate for possible system bus contention. For now, the buffer contains 256 locations for data. However, there is plenty of room for expansion should this prove to be insufficient.

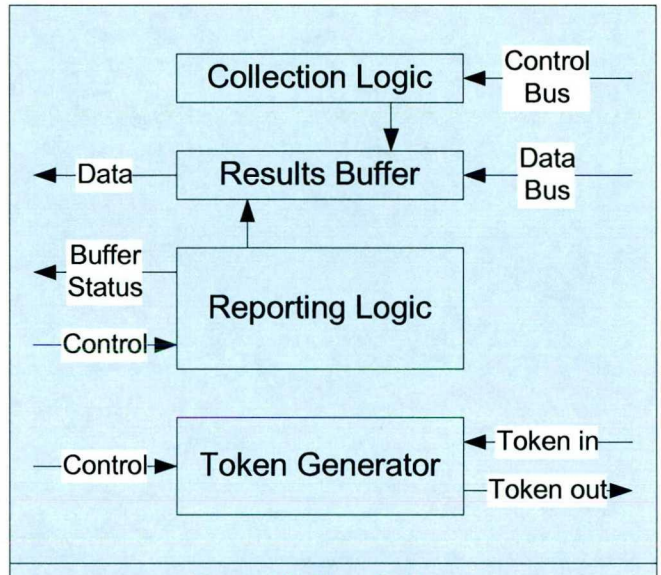

Figure 12: Data Collector

Again, the design goal was to keep the subcomponent as simple as possible while providing only the minimum necessary functionality. The single bus with an access token was used because it could be implemented independent of the number of PEs allowing for simple expansion and contraction of the system as needed for new hardware. Also, a circular buffer was used to simplify control logic. Though unlikely, it is possible for more results to be generated than the buffer can hold, should a prolonged period of external system bus utilization take place. Given the improbability of this taking place, and the further improbability of enough results building up to cause an overflow, it was decided that such a condition could simply be marked in the buffer itself to alert the user. In future work additional control lines will be added that will disable the reporting of data to the Data Collector in the event the Data Collectors buffer fills. This would essentially cause each PE to temporarily hold buffered data, greatly reducing the probability of an

overflow. In the event that an overflow did occur, it would corrupt less data (only a particular PEs buffer), and, further, that data would more than likely be useless anyhow, given the meaning of so many matches in such a short span of T.

System Controller: Main Controller

The main system controller is a collection of four sub-controllers that are responsible for interlinking the Cray modules and the rest of our coprocessor. Each of these sub-controllers has distinct tasks and interconnections with the other sub-controllers. Dividing functionality again reduces complexity of individual units, eases debugging and optimization, and allows for greater reusability of code overall.

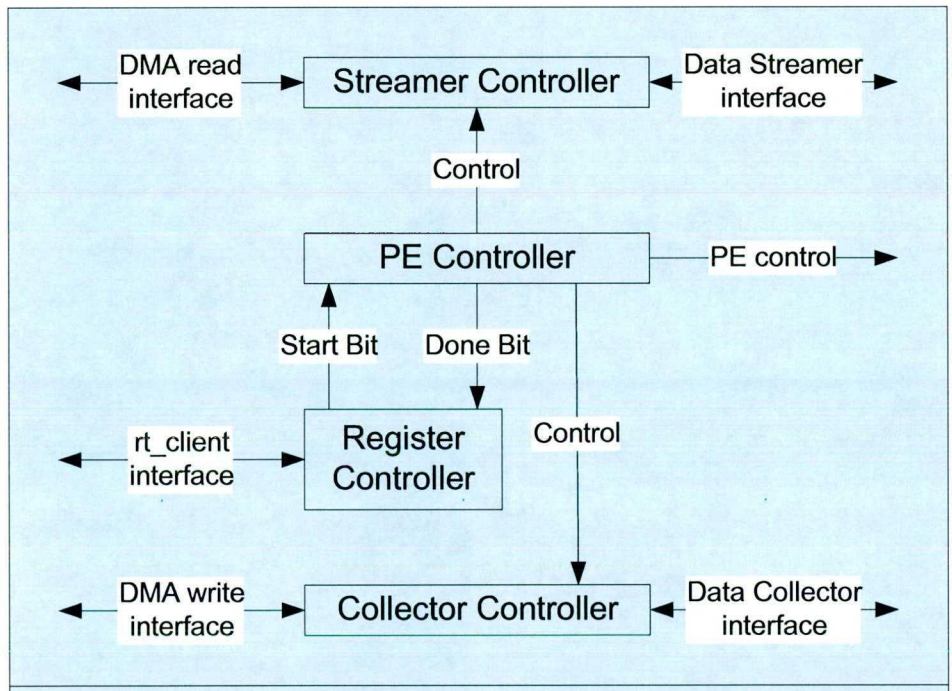

Figure 13: Main Controller

The first, and simplest, of these four sub-controllers is the register controller. This controller is an adaptation of some Cray example code. It creates a series of registers that are addressable both by the CPU and by a secondary interface on the controller. Further, it arbitrates read and write access to them between these two sources. This unit is essential in the communication of basic status information between the CPU and the coprocessor. The CPU uses a register in this controller to signal the coprocessor that all data is setup in memory, and that the coprocessor is free to begin execution.

The most complex sub-controller of all is the PE controller. This controller serves a variety of purposes. First, it contains a finite state machine that determines how it operates overall. During initialization of the machine, the PE controller is connected to the DMA read interface. Once it receives the signal from the register controller, it begins to read configuration data from the DMA memory space. It reads in a variety of information, including the number of Qs that need to be processed, the length of each Q, and the length of T. It also reads in all the Q values from this memory space and configures the PEs with them via the proper combination of control signals. Additionally, it initializes the Data Steamer and Collector. Throughout the entire execution, the PE controller asserts the needed control lines attached to the PE chain. Finally, it keeps track of the current position of the T string. Once T has been fully passed through the unit, it stops execution and calls for flushes of data out of all the relevant data buffers in the system.

The next sub-controller is the Streamer controller which interlinks the Data Streamer and the DMA read interface. After initialization, control of the DMA read interface is passed from the PE controller to the Streamer controller. Once this occurs, the Streamer controller ensures that the read buffer contained within the Cray DMA module fill and stays are full as possible. Whenever a request for another segment of T is received from the Data Streamer, this controller presents it with one from the DMA unit. It waits for the Data Streamer to latch the data into its buffer. The Streamer controller then waits for the Data Streamer to again signal for another segment.

The final sub-controller is the Collector controller which connects the Data Collector with the DMA write interface provided by Cray. Whenever the Data Collector has data in its buffer, it signals the Collector controller. Whenever receiving this signal, the Collector controller ensures that the DMA write interface is ready and, when it is, writes data from the Data Collector back into CPU memory space. This unit is also partially responsible for flushing the data from the systems' many buffers. It signals the Data Collector to perform a flush itself. It then writes all the data in the Data Collectors' buffer to the DMA write interface. Finally, it signals the DMA write interface to flush all data buffered within it to the CPU memory space. It then signals the PE controller that all flushes have completed.

Simulation and Verification

While designing the above described modules, a series of simulations and tests were used to ensure that at all stages of development the modules were working as intended, both within themselves and when interacting with other modules. In this way, top level testing, which is by far the most expensive in terms of time and effort, was minimized while overall confidence in the design was maximized. The following 5 steps formed the verification plan of the project:

1) Unit testing: whenever a module was created or modified, it was tested in isolation from all other modules. Simulated input tested either all classes of input or all classes of inputs whose results were last modified. Once the unit was working properly, testing finished.

2) Interaction testing: all interaction between modules were simulated thoroughly, two modules at a time. In this way, most glitches between output of one module and the input of another were captured and corrected without the added complexity of simulating other modules as well. This step usually resulted in additional unit testing of modules.

3) Data path testing: once all modules along a given data path were completed, for instance the PE controller, streamer controller, and data streamer, the modules were simulated together to ensure proper functionality from start to finish. In most cases, these simulations resulted in few errors due to the extensive previous testing.

4) Full unit testing: once all custom produced modules were completed and all data path testing yielded no errors, all the custom designed modules were simulated together. Cray provided module were not simulated, but rather emulated by input into the simulation. In this way the hardware was tested independent of Cray hardware.

5) Full system testing: in this case, all the modules that would be present in the FPGA were simulated to ensure that the Cray modules interfaced correctly with the custom written modules. This testing yielded only two errors over the course of the full development, primarily due to the extensive testing done prior to this final test.

CHAPTER 4

RESULTS AND PERFORMANCE ANALYSIS

Synthesis Results

After successful full system simulation, all the modules were translated from code into an FPGA configuration file, this process is referred to as hardware synthesis. This is a critical step in the design process, as the results of synthesis will tell both how fast the coprocessor will be able to run and how many processing elements the FPGA will accommodate at a time. The following table summarizes the synthesis results for all units outside the processing element chain.

Unit Name	Clock (MHz)	Slices
Controller	240	311
-Collector Controller	292	46
-Data Collector	274	57
-Data Streamer	333	44
-PE Controller	240	171
-Streamer Controller	292	60
Cray Core: dma_if	190	727
Cray Core: rt_client	270	468
Cray Core: reg_if	431	139

Table 3: Nonrecurring Module Synthesis Results

The total FPGA logic capacity usage by nonrecurring modules is estimated to be 1645 slices in addition to the space taken by the rt_core IP module. This is roughly 8% of the space on the FPGAs available on the Springfield XD1, which are Virtex2Pro50

chips with 23616 logic slices each. This leaves well over 20000 slices for occupation by processing elements. Note that the total number of slices taken by the controller is in fact less than the sum of slices taken by its components. This is due to optimizations built into the synthesizer that are working between components to remove redundant logic. Results of PE synthesis are listed below.

Unit Name	Clock (MHz)	Slices
Processing Element	257	305
-Dataflow Unit	344	119
-Popcount Unit	306	65
-Saver Unit	258	119

Table 4: Processing Element Synthesis Results

From these results each PE will take 305 slices. Note that in this instance, the sum of the slices used by the subcomponents is actually 2 slices less than the overall unit. This is caused by additional overhead needed for connecting the subcomponents together. For a safer estimate, logic usage is increased to 310 slices to allow for possible overhead, and thus the predicted capacity for a single FPGA will be approximately 64 processing elements. This number is likely to go up to 80 or even 100 PEs as further optimization of the PEs themselves takes place, as well as optimizations of other hardware present in the design.

During the course of synthesis itself, a number of warnings were observed. Most frequent among these were warnings about signals not being connected in component blocks. In all cases, the signals were found to be connected and, as no errors were found to be caused by this in the hardware, it is assumed that this warning is simply the synthesizer being overly cautious. Also, a number of warnings were observed reporting that signals had identical functionality. In these cases, the synthesizer removed the redundant signal automatically. Finally, a few errors were encountered that referred to specific register implementations within the design and possible optimizations. For all instances in the original SMART code, it was determined that these optimizations were not appropriate with regard to preserving functionality.

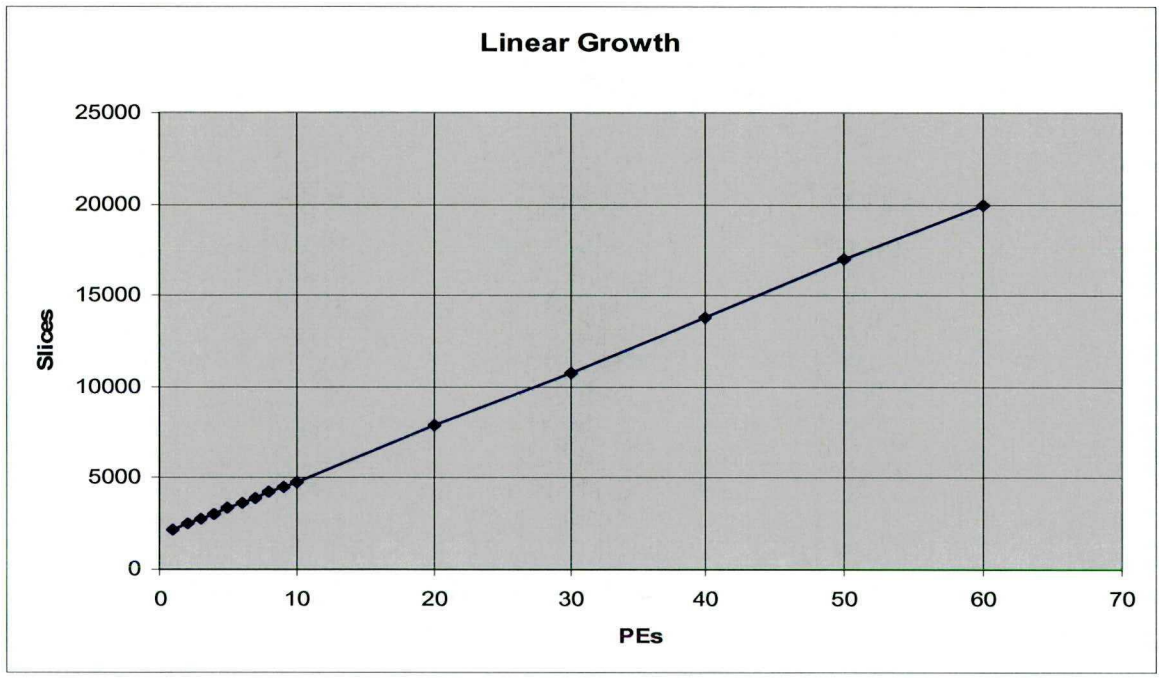

Figure 14: Linear Scalability

Also, synthesis reveals the scalability of the design. The chart below shows that the FPGA logic usage of the SMART coprocessor scales linearly for all explicitly synthesized configurations.

Once hardware was created, FPGA trials began and immediately a problem was uncovered. While the input test data should only have resulted in 5 matches being found, the coprocessor was reporting hundreds of matches. It was found that the number of erroneous matches reported scaled linearly with the input data and, likewise, the run time of the coprocessor. It was determined that the error was being caused by a signal line to the controllers' collector unit that was shared amongst all the processing elements. The synthesis of such a shared line created erroneous functionality of the unit without generation of an error or warning during synthesis. This shared line was replaced by a small amount of logic in the top level of the design that still scales linearly, and subsequently functionality in hardware was verified for 1, 5, 10, and 60 PEs (see Figure 14).

Performance

Prior to hardware testing, the following predictions about the SMART coprocessors' real world performance were made: first that the coprocessor will be able

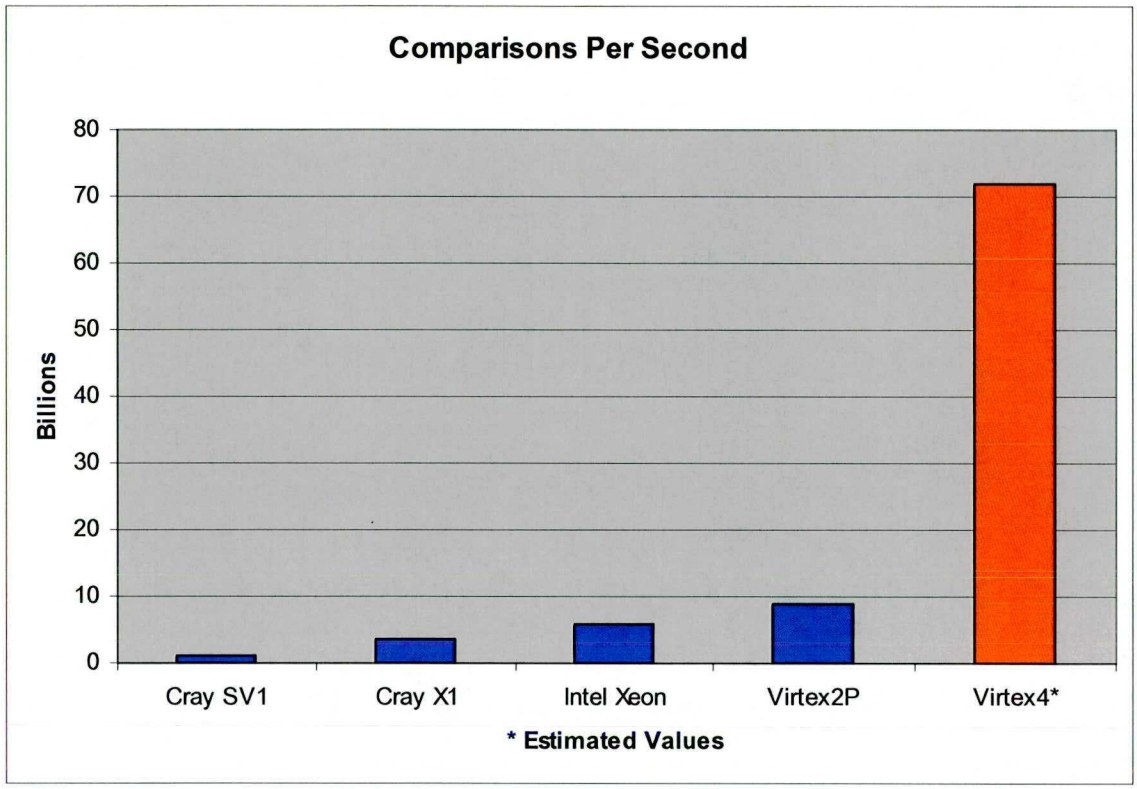

Figure 15: Raw Performance

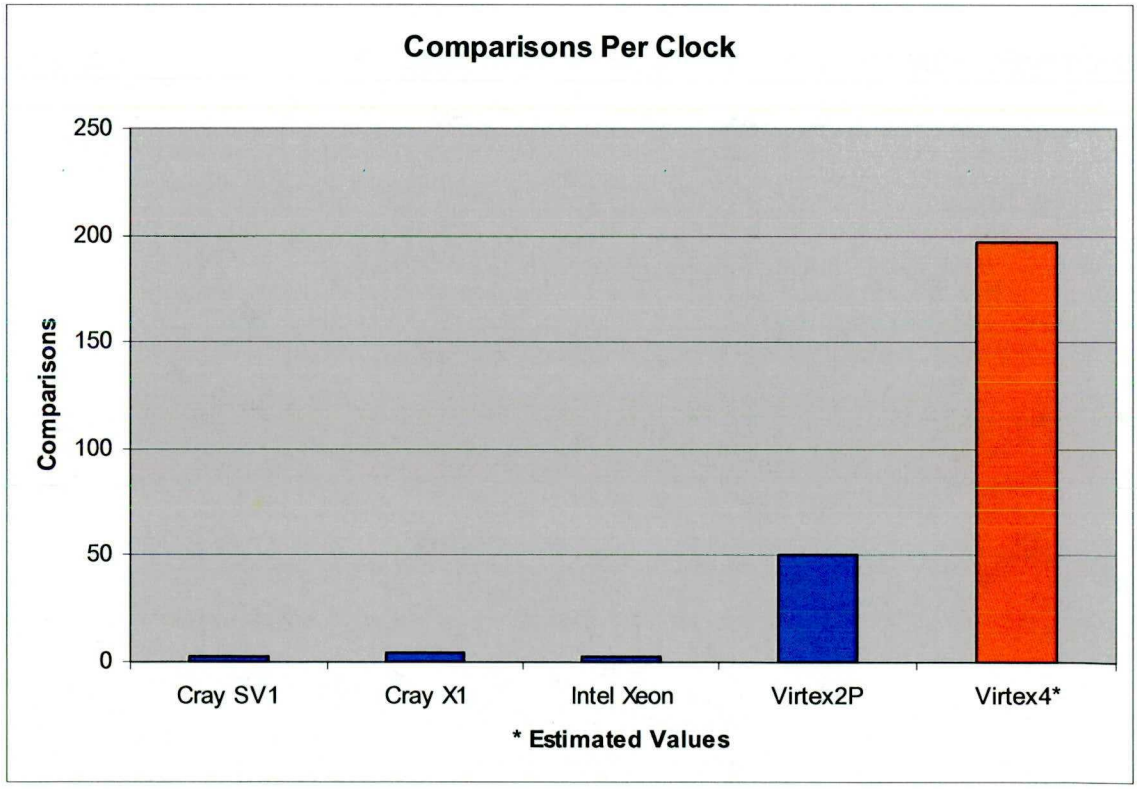

Figure 16: Clock Efficiency

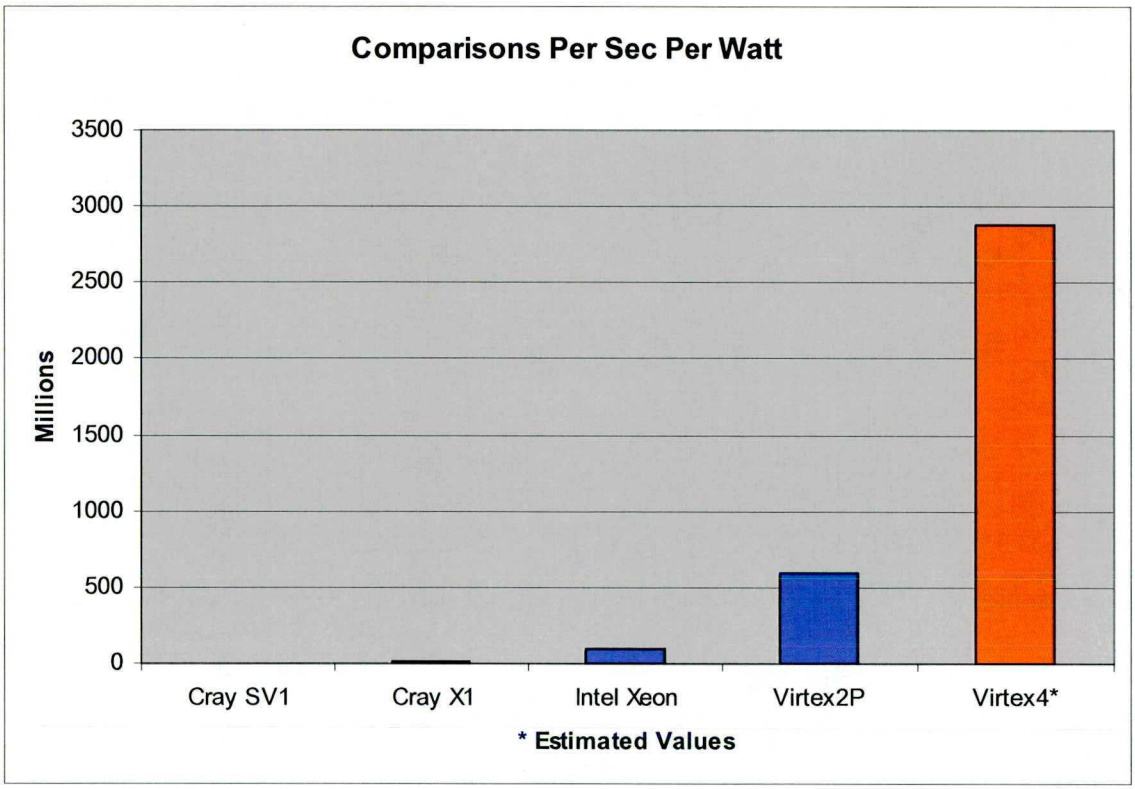

Figure 17: Power Efficiency

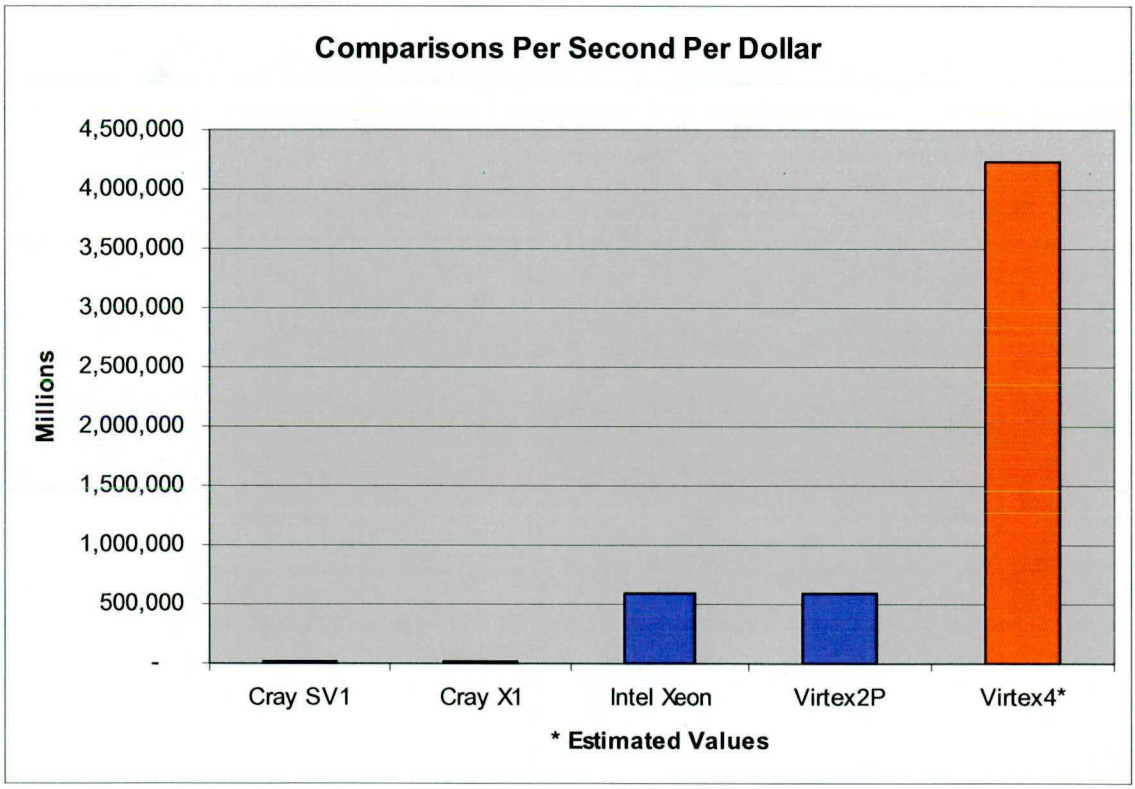

Figure 18: Cost Efficiency

to run at least 180MHz, possibly faster; second, that the SMART coprocessor will contain at least 60 processing elements. After implementation, the target of 180MHz was indeed attained. Currently the largest version of the SMART coprocessor contains 50PEs. With the current hardware code as many as 55 may be attainable, however with depredated clock performance. With additional optimization of the PEs it should be possible to attain 60 or even more PEs in the final design.

It was determined that a reasonable performance metric number of raw character comparisons per second. With this metric, the SMART coprocessor was predicted to achieve a theoretical performance rating of 345.6 billion comparisons per second and currently has achieved 303.2 billion. These numbers are obviously greater than actual performance when taking into consideration file I/O time, configuration overhead, and other factors. However, through intelligent software design, these effects can be largely minimized for large data inputs.

The following performance measures have been calculated. This set of measures fully expresses the power and advantage of reconfigurable computing. These performance metrics include comparisons per second, comparisons per clock cycle, comparisons per second per watt, and finally comparisons per second per dollar. For information regarding the development of these statistics, please refer to appendix A. First presented is a measure of raw performance, comparisons per second (see figure 15).

A reasonable measure of efficiency is Q-comparisons per clock cycle. As systems that get more done per clock cycle they tend to use less power, this measure will add insight to the next performance measure (see figure 16).

The next metric presented is comparisons per second per watt. This is an excellent measure of efficiency and can be a good predictor of cost of operation for a system. The wattage of processing units as well as comparisons per second per watt are noted on the chart below[28],[29] (see figure 17).

A final comparison metric presented here is one of economics. Present below is the Q-comparisons per second per thousand dollars of each of the seven solutions presented so far. The dollar figure used in this equation is based upon the cost of the portion of the system used. For the most part, this is simply a division of the full system cost by the percentage of the system utilized. However in the case of the SMART

coprocessor, a more fair measure is the percentage of the XD1 cost, if the XD1 were totally unequipped with FPGAs, plus the cost of a single FPGA expansion board. Therefore, this measure was estimated (see figure 18).

The SMART coprocessor shows itself to be extremely competitive, especially when considering metrics involving economic and efficiency factors.

CHAPTER 5

FUTURE WORK

Optimizations

With any system of this size, there are invariably optimizations that can and should be made. In implementing the initial design, the focus was on achieving a stable, working coprocessor. Now that this has been accomplished, optimization can begin. Most of the optimization effort will focus on the PEs, as any savings in terms of logical area will be multiplied along the enter PE chain. The following optimizations of the PE are planned in the near future:

Performance optimization of the Popcount unit could reduce its logical footprint by as much as 50%. The optimization would be accomplished by collapsing the 6 pipeline stages into 3, eliminating the intermediary 32x2 and 8x2 bit buffers. This modification would reduce the maximum clock speed of the unit, however experimentation has shown that it will most likely remain well over 200MHz

Performance optimization of the Data Flow unit could help reduce its logical requirements tremendously. This would be done primarily by using a small, single line block ram rather than a register to store the Q value. As this value changes very little, the extra power granted by a register is unused. Additionally, it may be possible to eliminate the T buffer by converting the top most level of the Popcount unit into a shifting register itself. Through these two optimizations, it is possible that the dataflow unit could be reduced in size by as much as 90%.

Finally, a design change may make it possible to remove the 32-bit counter from the Saver unit. This counter constitutes a large percentage of the logical requirement of this unit. Rather than have every PE contain a counter, it may be possible to move the

counter to the system controller and stream the counter value to the PEs along with control signals and T characters.

Outside of the PE, there are a few optimizations that are significant enough to be compelling. The first of these is the elimination of the write interface and associated logic of the Cray dma_if. This could increase overall system clock speed and free up a significant number of resources. To do this, the interface provided by the rt_core would be used directly when writing data back to the CPU, as write volume is lower and timing is less critical.

Also, functionality to implement a CPU interrupt already exits within the FPGA cores. By using this to signal completion of execution, rather than a status bit, the CPU would be free to perform other tasks while waiting for the SMART coprocessor to complete its task.

Functional Improvements and Additions

Most of the functional improvements planned involve the processing elements. First among these is the implementation of a mask in the PE that will allow Qs of lengths shorter than 32bp to be searched for. This is a simple matter of adding another stage to the top of the pipeline that ANDs together a mask and the result of the Q-to-T character comparison.

Another improvement would be the addition of additional control lines between the PEs and the data collector that would allow the data collector to deny the PEs access to the reporting bus. In this way, the data collector could prevent overflows in its buffer should it be unable to report results for an extended period of time. This may prove to be more costly than its value to the overall system however, due to the small number of matches reported during a run.

Another valuable improvement would be the addition of an ASCII to 2-bit binary encoder. This unit would compress incoming 8 character ASCII strings into 2-bit binary coded strings. This would remove a significant processing burden from the CPU in such a way as to be hardly noticeable, so long as transmission of data from the CPU to the FPGA can be sustained at this higher rate.

Finally, there are currently 64 bits in the main system status register, 32 that are processor read only and 32 that are FPGA read only. At present, only two of these are used to signal the start and end of computation. Additional functionality could be added to the remaining bits, including information such as coprocessor present state, buffer overflows, DMAs current location in memory, etc. This information could lead to a variety of other improvements, such as the ability to begin processing before all data is loaded into memory.

CHAPTER 6

CONCLUSION

SAGE processing has for many years been expanding in its capacity, detail, and scope. Large depositories of SAGE data have now been collected for a wide number of organisms, and wide-scale use of SAGE data is now assured. SMART is presented as a computational sophisticated solution. It should enable greater, more detailed analysis of SAGE data. SMART is implemented on a reconfigurable computing system to take advantage of the massive amount of parallelism inherent in SAGE data analysis.

Careful design ensures that SMART will be able to expand to meet future computational needs as well. Additionally, SMARTs design allows for greater flexibility when compared to other solutions and lends itself to other uses outside of SAGE data analysis. Additional functionality can be added with relative ease and, in doing so, future developers should only need to verify the PE elements themselves, rather than the full system. Addition of functionality should add only a negligible amount of resource usage and latency. Additionally, optimization of SMART is greatly eased by its highly repetitive structure.

Most impressive is the efficiency of the SMART processor. Not only does it match performance with the most advanced alternatives, it does so while using less power, at a lower clock cycle, on hardware that costs far less than the alternatives. The trade off for this performance is a long and difficult development cycle. However, as reconfigurable computing becomes more wide-spread, further work on development tools will ease these problems to the point where development is little more difficult than traditional software development.

Overall, the SMART coprocessor will be a useful tool in the analysis of genomic data, especially as databases of such data increase and searches become increasingly

more complex. Most pointedly, the flexibility, scalability, efficiency, and power of reconfigurable computing have been made clear. As FPGAs continue to grow in size and increase in speed, the power of this form of computing will continue to grow.

APPENDIX A

NOTES ON STATISTICS

Notes on Comparisons Per Second

Comparisons per second represents a raw number of character comparisons. For each Q string comparison to a substring of T, the number of comparisons per second generated is equal to the length of Q. For SAGESPY, Q is typically of length 21, while for SMART, Q is typically of length 32. It should be noted that both systems allow for variable length Q's, however these typical lengths were used for the generation of the relevant statistics.

Below, in tabular form, is a listing of comparisons per second:

SAGESPY, Cray SV1 – 1 Billion
SAGESPY, Cray X1– 3.7 Billion
SAGESPY, Intel Xeon – 5.9 Billion
SMART, Cray XD1 w/ Virtex2P – 288 Billion
SMART, Cray XD1 w/ Virtex4 (est) – 2336 Billion

Notes on Clock Speed

Clock speeds presented for the Virtex4 are an estimate, as actual hardware was unavailable.

Below, in tabular form, is a listing of clock speeds

Cray SV1 – 450MHz
Cray X1 – 800MHz
Intel Xeon – 2.2GHz
Virtex2P – 180MHz
Virtex4 (est) – 365MHz

Notes on Wattages

Wattages used for statistical generation include only the wattage of the actual processor involved in the computation. This was chosen for several reasons. First, determination of the wattage of a cluster is very difficult and would most likely require the installation of special monitoring equipment, and would result in machine down time. Second, including wattages for the full system is in many cases unfair, as some systems have high performance interconnects built into them, which consume large amounts of power, but are not used in computation. Likewise, FPGA systems have CPUs that

consume power but are only involved tangentially in computation. Take all these factors into account is difficult, if not impossible, so the best solution left was to simply use processing unit power comsumption.

Below, in tabular form, are the wattages used for each system

Cray SV1– 222W
Cray X1 – 203W
Intel Xeon – 60W
Virtex2P (est) – 15W
Virtex4 (est) – 25W

Notes on Cost

Costs are generalized system prices for systems located at the Ohio Supercomputer Center. It was decided not to attempt to adjust these costs for inflation, as no known depreciation method seemed a fair approach. Further, leaving the original prices shows the advancement of computation with time, which is still a beneficial metric.

As with wattage, there was difficulty in deciding how to measure cost. As the cost of the individual processor was not available in many instances, it was decided that the best method available to be to take into account the full system price, divided by the number of processors in the system. Note that the Intel Xeon cost is based upon an estimate cost of a single processor workstation. Also, note that the cost per node of the XD1 is greater than simple division would suggest. This is due to the fact that only 6 of 18 nodes are equipped with FGPAs, and the cost has been adjusted to account for this. Also, the price of the Virtex4 node is an estimate base upon the relative cost of the Virtex2P chips and the Virtex4 chips.

Below, in tabular form, are the number of processors in each system as well as a ballpark cost of each system.

Cray SV1 – 16, ~$1.5 Million, per node, approx $100K
Cray X1 – 12, ~$2 Million, per node, approx $150K
Intel Xeon – 1, ~$10K
Cray XD1 w/ Virtex2P – 18, ~$200K, per FPGA node, approx $15K
Cray XD1 w/ Virtex4 (est) – per FPGA node, approx $17K

BIBLIOGRAPHY

[1] "Complementary DNA sequencing: expressed sequence tags and human genome project." Adams MD, Kelley JM et al, Science, June 1991

[2] "FPGAs in Reconfigurable Computing Applications." Fawcett B, ISBN# 0-7803-2636-9

[3] "High-Level Language Abstraction for Reconfigurable Computing." Najjar W, et al. IEEE Computer Society, August 2003

[4] "Reconfigurable computing: architectures and design methods." Todman T J, et al, IEE Proceedings, 2005

[5] "Serial Analysis of Gene Expression." Velculescu VE, Zhang L, et al, Science, October 1995

[6] "Analysing uncharted transcriptomes" Velculescu VE, Vogelstein B, Kinzler KW, TIG, October 2000

[7] "Serial analysis of gene expression (SAGE): unraveling the bioinformatics tools." Tuteja R, Tuteja N, BioEssays 26:916-922, 2004

[8] "The Arabidopsis Root Transcriptome by Serial Analysis of Gene Expression. Gene Identification Using the Genome Sequence." Fizames C, et al, Plant Physiology, January 2004

[9] "Using the transcriptome to annotate the genome." Saha S, Sparks AB et al, Nature Biotechnology 20: 508, 2002

[10] "Robust-LongSAGE (RL-SAGE): A Substantially Improved LongSAGE Method for Gene Discovery and Transcriptome Analysis." Gowda M, Jantasuriyarat C, et al, Plant Physiology 134(3):890, 2004

[11] "Gene Expression Omnibus." http://www.ncbi.nlm.nih.gov/geo/

[12] "Saccharomyces Genome Database." http://www.yeastgenome.org/

[13] "GenBank" http://www.psc.edu/general/software/packages/genbank/genbank.html

[14] "Identification and prevention of a GC content bias in SAGE libraries." Marquilies E, Kardia S, Innis J, Nucleic Acids Research, 2001, Vol 29, No. 12 e60

[15] "Charactersization of the Yeast Transcriptome." Velculescu V, et al, Cell, Vol 88, 243-251, January 24, 1997

[16] "Identifying novel transcripts and novel genes in the human genome by using novel SAGE tags." Chen J, et al. PNAS, September, 2002, vol 99 no 19, 12257-12262

[17] "SAGE is far more sensitive than EST for detecting low-abundance transcripts." Sun M, et al. BMC Genomics, January, 2004

[18] "A Comparative Molecular Analysis of Developing Mouse Forelimbs and Hindlimbs Using Serial Analysis of Gene Expression (SAGE)." Margulies E, Kardia S, Innis J, Genome Research, June 2001

[19] "Computational Analysis of Gene Identification with SAGE." Clark T, et al. Journal of Computational Biology, Vol 9, Num 3, 2002, 513-526

[20] "Bioconductor: open software development for computational biology and bioinformatics." Gentleman R, et al. Genome Biology 2004, 5:R80

[21] "Reconfigurable Architectures for Bio-Sequence Database Scanning on FPGAs." Oliver T, Schmidt B, Maskell D. IEEE Transactions on circuits and systems-II: Express Breifs, Vol 52, No 12, December 2005

[22] "Reconfigurable Computing Systems." Bondalapati K, Prasanna V. Proceedings of the IEEE, Vol 90, No 7, July 2002

[24] "Identification of common molecular subsequences." Smith T F, Waterman M S. Journal of Molecular Biology, vol 147, 195-197, 1981

[25] "Basic Local Alignment Search Tool." Altschul S F, et al. Journal for Molecular Biology, vol 215, 403-410, May 1990

[26] "High Performance Genome Scale Comparisons for the SAGE Method Utilizing Cray Bioinformatics Library (CBL) Primitives." Gowda M, Wang G, et al., CUG'05 proceedings, 2005

[27] "Cray Bioinformatics Library (CBL) Datasheet." Cray Inc. BioLib Release 2.0, 2003

[28] "Cray XD1 Site Planning", Cray Inc, 2004

[29] "Cray X1 Site Planning", Cray Inc, 2003