Development of the SMART Coprocessor for Fuzzy Matching in Bioinformatics Applications

By Harrison B. Smith and Eric A. Stahlberg
The Ohio Supercomputer Center, Columbus, Ohio

ABSTRACT: Efficiently matching or locating small nucleotide sequences on large genomes is a critical step in the Serial Analysis of Gene Expression method. This challenge is made increasingly difficult as a result of experimental assignment errors introduced in the match and target sequence data. Early versions of SAGESpy were developed incorporating the Cray Bioinformatics Libraries to enable large scale pattern matching of this type. This presentation describes the design of an FPGA-based scalable fuzzy DNA sequence matching algorithm implemented specifically for the XD1. Results of this implementation will be compared to performance on earlier Cray SV1 and current Cray X1 systems.

KEYWORDS: SAGE, FPGA, CBL, fuzzy match, XD1, SAGESpy, bioinformatics

Introduction

Understanding the behavior of a complex biological system is a daunting prospect. Even with the availability of a well characterized genome, the defining genetic program for the organism, tremendous amounts of effort remain to characterize the overall behavior of these extremely complex systems. During the lifetime of the organism, different genes become active, or are expressed, in response to the natural aging of the cell and in response to stress including disease. The qualitative identification of the expressed genes combined with the quantitative measurement of levels at which these genes are expressed over time provides the insight into the behavior and response of the biological system.

Several experimental methods have been developed to study the expression levels of genes. Sequencing of cDNA clones has enabled identification of the expressed genes (Adams et al 1995). Subtractive hybridization and differential display (Hedrick et al 1984, Liang and Pardee 1992) have proved useful in identifying differences for sufficiently expressed genes. Arrays of large numbers of DNA oligonucleotides are also used to simultaneously compare the expression of thousands of genes (Lockhart et al 1996, Mahadevappa and Warrington 1999). Unfortunately these methods are limited to characterizing known genes and do not provide insight for uncharacterized sections of an organisms genome.

The serial analysis of gene expression (SAGE) method (Velculescu et al 1995) has been developed to simultaneously analyze thousands of transcripts, including previously uncharacterized transcripts. The SAGE method was developed to study expression levels in multiple states of human pancreatic cells, including normal, developing and diseased. Over time, the SAGE method has been found to be a suitable method to study the effect of drugs on tissues, identification of disease related genes, characterization of disease pathways and investigations of other organisms (Madden et al 2000; Saha et al 2002; Chen et al 2002, Fizames et al 2004). The SAGE method is based on two principals, the use of a short nucleotide sequence 'tag' containing sufficient information to discriminate the transcript, and concatenation of these tags within a single clone to enable efficient sequencing. The concatenation of multiple tags into a combined clone, with a characteristic sequence serving as a sentinel separator, enables the capture and subsequent sequencing and characterization of genes with even low expression levels. Originally defined for short sequence tags of 9-10 nucleotides, further enhancements of the SAGE method have been developed using longer sequence tags with stronger discriminating power. The Long Sage (Saha et al 2002) and RL-SAGE (Gowda et al 2004) extend the length of employed SAGE tags to 21 nucleotides.

An organism genome consists of several chromosomes, each comprised of DNA. The DNA in turn is comprised of linked nucleotides, substances cytosine, adenine, thyamine and guanine, connected in a double-helix arrangement on a phosphate sugar framework. (These sequences are frequently represented as strings comprised of the letters CATG, representing the individual nucleotides.) Both the sequencing of the experimental cDNA tags and the identification of candidate tags from the organism genome, defines strings of characters containing arrangements of letters CATG to be used for subsequent comparison.

To identify and quantify the tags expressed in the cell, the cDNA sequence is analyzed for both tag composition and tag population. These tags are identified by comparing the character sequences for the experimental tag (query) to character sequences for the candidate tags (targets) derived from the organism genome, and associating these to specific genes or functional areas. Regularly, lists of thousands of experimentally characterized tags are compared against tens to hundreds of thousands of tags prepared from an organism genome. In more complex genomes, such as is the case for maize, the numbers can reach hundreds of thousands for each of the experimental and candidate tags. Adding to the complexity of the comparison is the need to account for sequencing errors, mis-assigned nucleotides, in both the experimental sequence tags and organism genome. It is this last requirement, the need to account for sequencing errors, which precludes the use of a basic substring search algorithm and necessitates the need for a 'fuzzy' matching algorithm.

The SAGESpy application (Gowda et al 2004) was developed specifically for SAGE analysis. The primary motivations for developing the SAGESpy application were computational efficiency and researcher flexibility. The common BLAST application, in addition to being computationally inefficient for SAGE analysis, also requires pre-staging of target sequences. Alternatives to BLAST, such as Smith-Waterman, while viable, would add unnecessary extra operations and subsequent computational cost of the analysis.

Methodology

The SAGESpy application was developed using the Cray Bioinformatics Library (CBL) and has shown good performance on the Cray SV1 and X1 platforms (Gowda et al 2004). The development of the SMART coprocessor is intended to provide a high-performance replacement for the cbsearchn() routine which is at the core of the SAGESpy application analysis.

SAGE analysis of genetic material generates a large volume of textual data. The strings generated by SAGE analysis represent specific base pair (bp) sequences located within the analysis. Results and are stored in FASTA format, with short character sequences comprised of characters A, T, C, and G each representing specific nucleotides. In processing the data from SAGE, fuzzy matching operations take place between these various strings. Specifically, short, predefined strings are searched for in the longer SAGE generated strings. The shorter strings will be referred to as query strings, or Q's. The longer string being searched will be referred to as the target, or T.

This fuzzy matching can be accomplished in several ways. The first is a brute-force permutation method wherein, a query search string (Q) is permuted into a series of all acceptably similar strings. All of these permuted Q's are then searched for within a target sequence (see Figure 1).

In the case of SAGE analysis, allowing for a single error in a sequence of length n, this would result in an *O(n)* operation per match.

Allowing for two errors in a sequence increases this to an $O(n^2)$ operation per match, three errors increases this to $O(n^3)$, and so on. The rate of increase in the number of acceptable strings makes anything more than single errors impractical except for exceedingly short search and target strings.

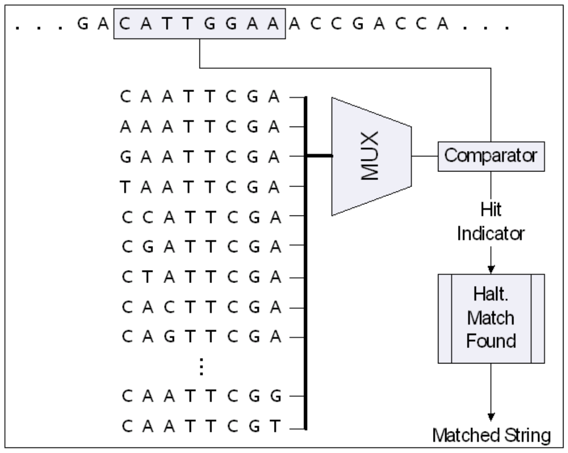

Figure 1. Fuzzy matching via permutation.

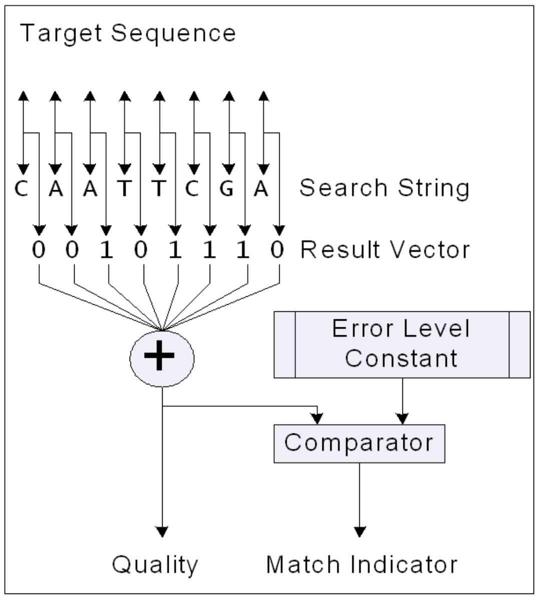

Figure 2. Fuzzy matching with popcount

The second method for fuzzy matching is the population count or *popcount* method. In this method the search string is compared to a substring of the target. Each element of the search string is compared to a corresponding element in the target substring. This comparison generates a bit array. Within the bit array, a match between elements is represented by a 0 and a mismatch by a 1. This bit vector is then summed and if the sum is sufficiently small, it is considered a match (see Figure 2). This results in n comparisons for vectors of length n, followed by a summation of n elements. Note that the computational complexity of this method is $O(n)$ independent of the allowed number of mismatches.

A third possibility involves the use of a finite state machine (FSM). In this method, a hardware machine keeps track of the incoming T (target) sequence and compares it serially to the stored Q (query) sequence. This opens up possibilities for allowing for omission and additions to the fuzziness of the match. However, computation time for an FSM implementation would be variable and require many concurrent FSMs to achieve throughput of one sequence per clock cycle. In the case of additional string variability, a processor based on the Smith-Waterman algorithm would be expected to provide a better solution allowing additional fuzziness in the match while maintaining predictable runtime.

Of the three approaches, the second approach remains most optimal for the SAGE analysis on a reconfigurable platform. The first method presented is the brute-force approach and is computationally burdensome and inefficient. Yet, being a brute-force approach it may remain the only option for particular systems. The third method has the potential to be quite powerful, but is also increasingly complex, require additional gate resources, and lead to overall poorer net performance for the simple fuzzy match. The second approach, using the SMART processor, provides for a scalable parallel design suited for reconfigurable systems.

Implementation & Design

The acceleration system is being implemented on the Cray XD1 at the Ohio Supercomputer Center (OSC), Springfield site. The system at OSC Springfield (OSC-S) has Xilinx Virtex2Pro50 FPGAs attached to six of eighteen nodes. Cray provides a variety of hardware modules for the attached FPGA that allow for various functionality, including communication between the CPU and FPGA, decoding requests from the communication interface, allowing access to the FPGA attached QDRII memory system, allowing direct access to the CPU memory via DMA, and implementing a CPU interrupt. These provided modules take care of some of the most difficult CPU to FPGA interfacing challenges, allowing the application developer to focus on the implementation of the actual solution to the problem. However, at the same time, use of these modules ties an implementation explicitly to the XD1 system. The Cray XD1 also has installed many Cray HPC libraries, including the CBL (Cray BioLib).

Since the analysis of data is typically comparing thousands (of even hundreds of thousands) of Q's to a single T, the application lends itself to parallelization with great ease. The Cray XD1 provided several important advantages for this design. The cluster structure of the XD1 will easily accommodate high-level parallelization. Secondly, the available CBL provides an algorithm to both read the FASTA file format and compress the data to 2-bit values. This reduces the amount of memory usage in the comparison process by a factor of four, and will reduce the required bandwidth to move data through the accelerator.

Full System Overview

The high level structure of the acceleration system is that of a serial string of custom designed popcount processors controlled by a single controller tied into the Cray provided hardware. The accelerator can accommodate multiple Q's, dependent only upon the size of the FPGA provided with the XD1. Theoretically search rates of 180 million bps per second for 60-80 Q's can be attained with the six FPGAs currently part of the OSC-S XD1 system.

The serial string of popcount processors design has several significant advantages. The popcount processor is reasonably compact, highly efficient, and adapts well to a serial pipeline structure. The pipeline structure itself works well because of both the embarrassingly parallel nature of the problem and the typical extent of T being many times larger than the number of Qs, meaning overhead to fill the pipeline is negligible relative to overall performance. This allows a large number of Q's to be compared to the same T while only needing to stream the T through the system minimally. Additionally, the serial nature of the PE chain reduces fanout and eliminates any need for large, slow decoders.

Processing Element

The workhorse of the accelerator is the long chain of processing elements (Figure 3) that will take up the majority of FPGAs real estate. Each PE in the chain passes along almost all the input it receives from either the previous PE or the system controller. There is a wide bus to allow for the rapid passing of Q values, a number of streamed control signals, and a serial bus for the streaming of the T values. The PE does one comparison per clock cycle by taking advantage of a six stage deep computational pipeline. Each PE holds one 32 bp Q value and 32 bp of the T substring. These values are fed into the fuzzy matching computational pipeline, and each clock cycle the 32 bp of the T substring are shifted. Each PE has a small amount of dedicated memory within which it buffers the positions of the matches it finds. These results are transmitted to the system controller over a bus shared by all the PEs.

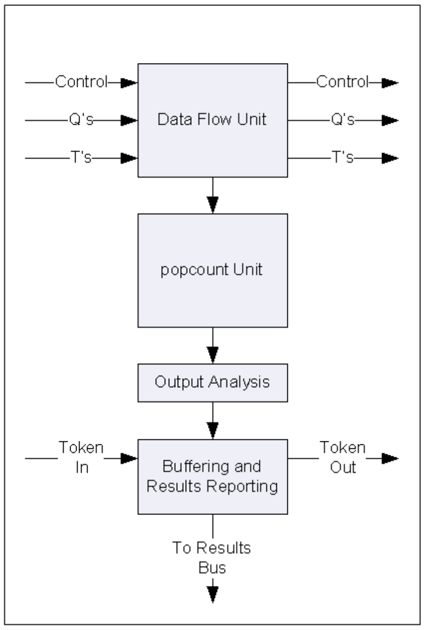

Figure 3. Processor element design.

The decisions contributing to the design of the processing element yielded many advantages that increase performance, decrease the FPGA real estate required, and decrease communication overhead. The primary decision was to have the processors act as a serial chain with pipeline like features. Because the length of the T string is typically in the millions of bps, overhead becomes a negligible portion of the overall comparison. Additionally, the approach allowed us to optimally design a single PE which would be replicated within the overall design. Due to the pipeline like nature of the chain, getting one PE to run at full speed is enough ensures optimal performance designs for all PEs. This also simplifies expansion of the system, allowing replication up to 1024 processors per FPGA. This limitation could easily be loosened with little extra work. Finally, the chain lends itself nicely to the bus based communication setup between the PEs and the main system controller.

Early in the design, a bus dedicated to the transfer of Q's was added to the PE. This was done to help simplify PE logic as well as the main system controller logic, and reduce overhead of the full system. Additionally, rather than have bps from T shifted all the way through the 32bp buffer within each PE, they are shifted back out again immediately as well as down the buffer. In this way, all inputs to a processing element are outputs to the same PE the next clock cycle.

The actual fuzzy matching engine is also deeply pipelined. The first stage compares Q to the T substring and generates a bit vector of length 32. The subsequent stages sum this bit vector and generate control outputs that indicate if the result should be buffered. Experimentation has shown that by collapsing some of the pipeline stages, resource usage goes down without effecting theoretical performance. Plans are to add an earlier stage to the pipeline that can mask off a specific bit range, allowing for the searching of strings of less than 32bp.

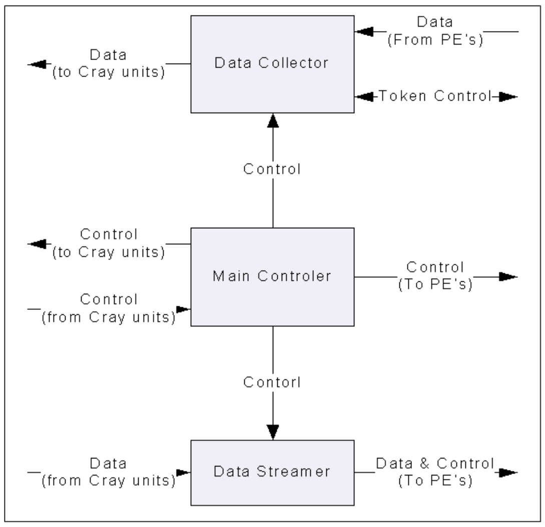

Figure 4. System controller design schematic

Another optimization made was the small buffers in the PEs. Given the relatively short length of search strings and the threshold level of the fuzzy matching allowing for only 2 errors, the statistical frequency of matches is extremely low. This allows the buffers to be relatively small when compared to the available block RAM built into the FPGA. Each PE was given a circular buffer capable of storing 128 matches. A design decision was made to simply note buffer overflow and continue processing in the event a buffer becomes overfilled. The indication of overflow allows the application developer to respond accordingly. Also, the relatively low match rate enabled use of a single bus to connect the controller to all PEs. Control of the bus is granted by a token originated at the system controller, and passed between the PEs. This allows for PEs to further reduce their dependence on their buffers by providing a low-overhead means to empty buffers.

System Controller

The most complex component of the fuzzy matching system is the main system controller (Figure 4). This controller consolidates the interfaces with both the portably designed PEs and Cray specific modules employed. The controller is responsible for providing configuration, a constant stream of input data, as well as collecting results from the PEs during the course of execution. 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, interfacing via the hardware modules provided by Cray. The controller makes use of several Cray modules to achieve these objectives. The system controller is composed of three subsystems each of which handle a specific function. These three subsystems are the Data Collector, the Data Streamer, and the main Controller. These three components will be discussed in detail later.

In designing the system controller, Cray modules were critical. Use of the Cray provided modules ensured us of properly working, highly tested components. Nearly as important, these components implemented the most challenging hardware elements in the design in an efficient and general way. They are highly reliable, easy to use, and do not present a bottleneck in the system. All communication and routing systems were essentially handled by Cray modules. We also divided the system controller into three subcomponents to help simplify and compartmentalize the design. Each subcomponent could be independently optimized, debugged, and verified which helped simplify and speed up development.

System Controller: Data Streamer

The Data Streamer unit (Figure 5) is a smart serializer that keeps the PEs supplied with elements of the target sequence at a rate of one bp per clock cycle. It relies on the Main Controller for parallel T data retrieved from the CPUs memory space. Its *smartness* comes from its awareness data. Rather than a simple shift register, the Data Streamer tracks where valid and invalid data are within its 32 bp buffer. The Data Streamer commences requesting data from the Main Controller when less than half the buffer is valid. Should the buffer ever empty completely, 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 5-bit counter. As the buffer empties, the counter counts down. Some simple logic makes up the rest of the unit, asserting and de-asserting lines as needed.

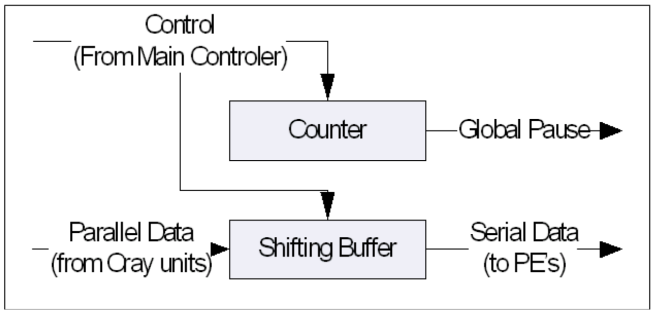

Figure 5. Data Streamer Design Schematic

System Controller: Data Collector

The Data Collector (Figure 6) is a simple token generator and a circular data buffer system. It creates the initial token that is passed between the PEs and monitors the bus connecting the PEs for data. When data is present on the bus, the Data Collector records the data into its own data buffer. The Data Collector keeps the Main Controller informed whenever it has any collected data and passes on data whenever the Main Controller requests it.

As before, 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. A circular buffer was used to simplify control logic, where the total quantity of reported matches would not be limited by buffer size. The Data Collector will signal to the Main Controller whenever it has any data to keep the buffer as empty as possible. In all probability the Data Collector will keep both the buffers in the PEs and its own buffer from ever overflowing for most encountered situations.

However, this is of course dependent on both the number of PEs and the data sets processed. It was decided, as with the PEs, that in any instance of the buffer overflowing it was sufficient to simply signal the user.

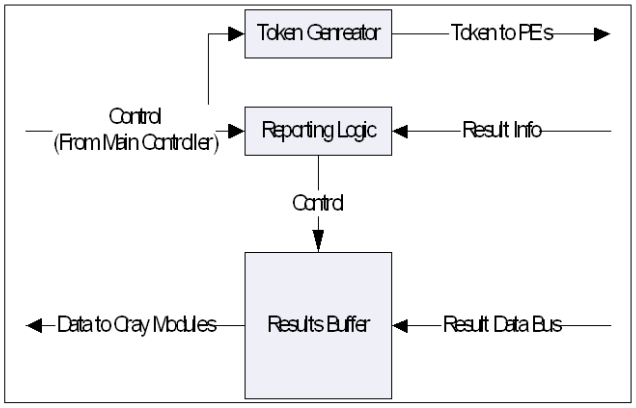

Figure 6: Data collector design schematic

System Controller: Main Controller

The main system controller (Figure 8) is a complex collection of sub-controllers that is responsible for interlinking the Cray modules and our custom modules. There is a controller that keeps the read request buffer of the Cray DMA module full at all times. The second of these subsystems uses data from the DMA read interface to keep the Data Streamer supplied with the bps of T. Another controller takes data from the Data Collector and uses it to make DMA write requests. There are two controllers that handle the reception of configuration data from the rt_client module. A final controller configures and coordinates the actions of the PEs and other sub-controllers.

The division of the Main Controller into a series of sub-controllers again served the purpose of helping to simplify overall design and achieve maximum performance. The DMA read controller ensures that as much data as possible is transferred to the FPGA before it is needed. This should keep the FPGA well supplied with data and help to avoid the need for system pauses. The controller connecting the Data Collector and the DMA write interface works to keep the Data Collector buffer empty as much as possible. This helps to further reduce the already small probability of and buffer overflows.

Performance

Having completed all of the above mentioned modules, full system testing and evaluation have begun. We are confident that all modules will be able to run at the highest possible clock speed allowed by the current FPGA, 200MHz. We include below a table of predicted clock speeds and area usages of modules. Each PE will process one bp of the target sequence per clock cycle. This leads to a predicted performance of 200 million bp every second. We also predict a final version of the accelerator will include between 80 and 100 PEs. This yields an estimated processing power of 20 billion bp per second. Shortly, the Virtex4 will be available. This FPGA has twice the clock speed, and 4 times the area of the Virtex2Pro FPGA. The increased capability of the Virtex4 will raise predicted performance levels to 160 billion bp per second. These predicted performances are very competitive with current systems as can be seen in Table 1. Performance data in Table 1 was taken from work done previously at the Ohio Supercomputer Center (Gowda et al 2004). Our design estimates are included for the current XD1 equipped with the Virtex2Pro and the theoretical scaling were the system equipped with a Virtex4.

Performance listings for the design do not include an estimate for I/O.

While very attractive performance is achieved, further work remains to increase overall performance. First and foremost, the fuzzy matching pipeline can be collapsed slightly. This will not increase clock speed but will serve to reduce the area each PE requires on the FPGA enabling more PEs per given FPGA.

Another modification would be the addition of a second comparator and popcount unit to the PE to allow for both forward and backward searching in the target string to take place simultaneously, reducing the need to stream T data through in both its direct and reverse complement form. Lastly, compression of the ASCII sequence data could be moved from CPU computation to onboard the FPGA. This could potentially save a large amount computation time on the CPU side by adding one more stage in the pipeline of the processing accelerator, effectively adding no time to execution but with a trade-off in I/O demands.

Conclusions

This project further illustrates an advantage FPGA hardware has over traditional hardware for processing bioinformatics data. Specialized hardware available on commodity processors today, such as floating point units, vector units, generally do not contribute to problems involving sequence comparison. The impact and potential value of reconfigurable computing hardware in these situations is emphasized not only by our predicted performance results, but also by the fact that such performance could be achieved on a modern desktop computer with a specialized FPGA board at a fraction of the cost and energy usage of a modern supercomputer.

About the authors:

Harrison (Ben) Smith is an Computer Science - Electrical Engineering graduate student at the Ohio State University in Columbus, Ohio. Ben may be reached at bsmith@osc.edu.

Eric Stahlberg is a senior systems manager a the Ohio Supercomputer Center (Columbus, Ohio) leading efforts in bioinformatics and FPGA application development. Eric may be reached at eas@osc.edu.

Both Ben and Eric may be reached at the Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, OH, 43212

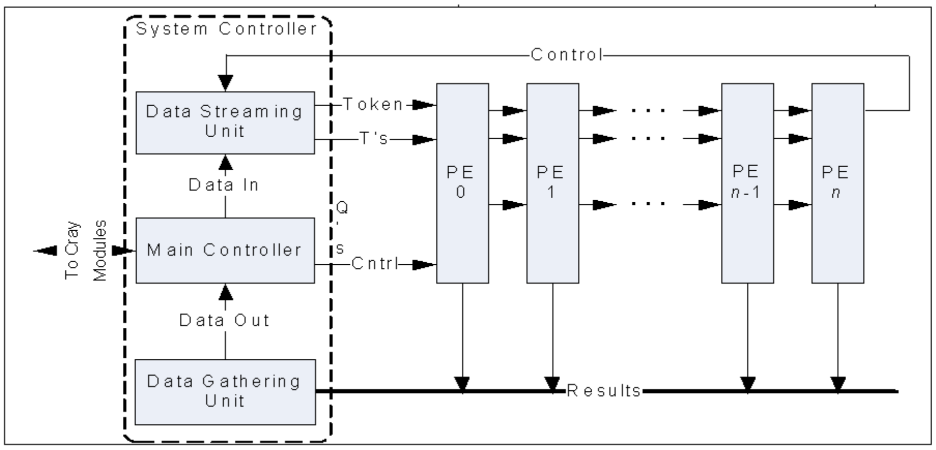

Figure 7. System design schematic

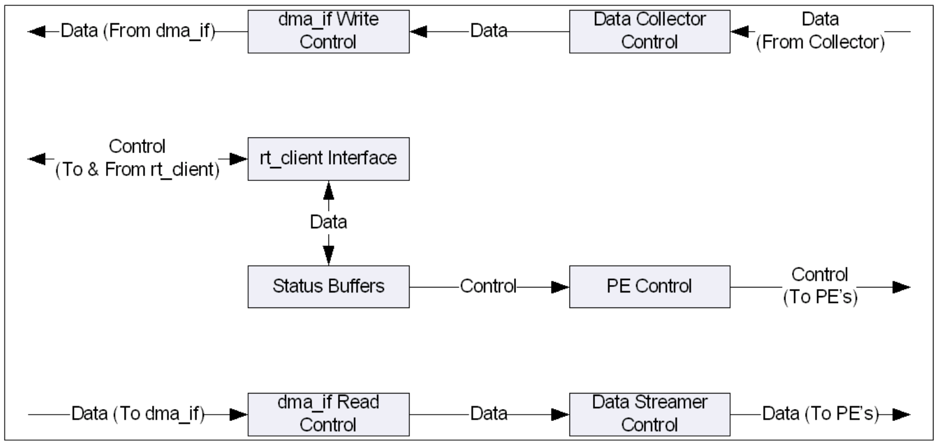

Figure 8. Main controller design schematic.

	Number of target bps	Computation time in seconds				Number of Hits
		SV1	X1	XD1 w V2P (estimated)	XD1 w V4 (theoretical)	
Chrom1	13333992	1.35	0.36	0.066	0.0165	29
Chrom4	9923256	1.00	0.27	0.049	0.0124	34
Chrom10	5964420	0.61	0.16	0.030	0.0074	8
TIGR(EST)	9889404	1.00	0.26	0.049	0.0124	333

Table 1. Performance of SAGESpy XOR comparison approach relative to estimates for SMART co-processor. Speed was measured for one hundred query tags with a two mismatch threshold.

References

Adams MD, Kerlavage AR, Fleischmann RD, Fuldner RA, Bult CJ (1995), "Initial assessment of human gene diversity and expression patterns based on 83 million nucleotides of cDNA sequence", 1995, Nature 377(Supplement):.3

Chen J, Sun M, Lee S, Zhou G, Rowley JD, Wang SM (2002), "Identifying novel transcripts and novel genes in the human genome by using novel SAGE tags", Proceedings of the National Academy of Science USA 99: 12257

Fizames C, Munos S, Cazettes C, Nacry P, Boucherez J, Gaymard F, Piquemal D, Delorme V, Commes T, Doumas P, Cooke R, Marti J, Sentenac H, and Gojon A (2004), "The Arabidopsis Root Transcriptome by Serial Analysis of Gene Expression. Gene Identification Using the Genome Sequence. Plant Physiology", 134(1):67.

Gowda M, Wang GL, Doak J, Manikantan S, Stahlberg E (2004), "High Performance Genome Scale Comparisons for the SAGE Method Utilizing Cray Bioinformatics Library (CBL) Primitives", Proceedings of Cray User Group 2005

Gowda M, Jantasuriyarat C, Dean RA, Wang GL (2004), "Robust-LongSAGE (RL-SAGE): A Substantially Improved LongSAGE Method for Gene Discovery and Transcriptome Analysis", Plant Physiology 134(3):890

Hedrick SM, Cohen DI, Nielsen EA, Davis MM (1984), "Isolation of cDNA clones encoding T-cell specific membrane-associated proteins", Nature 308:149

Liang P and Pardee AB (1992), "Differential display of eukaryotic messenger RNA by means of the polymerase chain reaction", Science 257:967

Lockhart DJ, Dong H, Byrne MC, Follettie MT, Gallo MV, et al. (1996), "Expression monitoring by hybridization of high-density oligonucleotide arrays", Nature Biotechnology 14:1675

Madden SL, Wang CJ, Landes G (2000), "Serial analysis of gene expression: from gene discovery to target identification", Drug Discovery Today 5:415

Mahadevappa M, Warrington JA (1999), "A high density probe array sample preparation method using 10- to 100-fold fewer cells", Nature Biotechnology 17:1134

Saha S, Sparks AB, Rago C, Akmaev V, Wang CJ, Vogelstein B, Kinzler KW (2002), "Using the transcriptome to annotate the genome", Nature Biotechnology 20: 508

Veluculescu VE, Zhang L, Vogelstein B, Kinzler KW (1995), "Serial Analysis of Gene Expression", Science 270: 484