RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation*

Krishna Muriki
ECE Department
Clemson University
Clemson, SC 29634–0915
kmuriki@ces.clemson.edu

Keith D. Underwood†
Sandia National Labs
P.O. Box 5800
MS-1110
Albuquerque, NM 87187-1110
kdunder@sandia.gov

Ron Sass
ITTC
University of Kansas
106 Nichols Hall
Lawrence, KS 66045–7523
rsass@ittc.ku.edu

Abstract

Basic Local Alignment Search Tool (BLAST) is a standard computer application that molecular biologists use to search for sequence similarity in genomic databases. This report describes the implementation of an FPGA-based hardware implementation designed to accelerate the BLAST algorithm. FPGA-based custom computing machines, more widely known as Reconfigurable Computing, are supported by a number of vendors and the basic cost of FPGA hardware is dramatically decreasing. Hence, the main objective of this project is to explore the feasibility of using this new technology to realize a portable, Open Source FPGA-based accelerator for the BLAST Algorithm. The present design is targeted to an ***AceIIcard*** *and the design is based on the latest version of BLAST available from NCBI. Since the entire application does not fit in hardware, a profile study was conducted that identifies the computationally intensive part of BLAST. An FPGA hardware component has been designed and implemented for this critical segment. The portability and cost-effectiveness of the design are discussed.*

1. Introduction

Scientists investigating biology at the molecular level are making great advances that have a significant impact on society. From understanding life [11] to renewable bioenergy [10] to the treatment of disease [4] to bioremediation [10], understanding how the cell works, this science holds enormous promise. Much of this work has come to rely on computers and information technology — not just to organize and catalog the volumes of data generated but also as a tool for discovery. Like the other sciences, molecular biology has come to embrace computation and computational techniques as a third branch of science.

The field, called bioinformatics or sometimes computational biology, is generally concerned with the discovery of new knowledge via information processing. This ranges from simulations of complex biological systems to genome sequencing and similarity searches. Here, we focus on the computational aspects of the latter. By improving a fundamental tool, BLAST [1], our goal is to increase the productivity of scientists aiming to determine the sequence of organisms' genomes, reveal gene function, uncover relationships between organisms, and other related activities. For example, one approach to the assembly process (a very computationally demanding step in the sequencing an organism's genome) uses BLAST repeatedly to find a solution. Hence, improving BLAST positively impacts a broader range of bioinformatic tools.

The BLAST algorithm is a heuristic. Given a subject database and a query (both of which are sequences of some alphabet,[1]) the basic idea is to find positions in the subject where the query is similar. By similar, we mean that the query matches, letter for letter, permitting a number of variances (as specified by a command-line parameter). Allowable variances include situations such as omitted letters, added letters, and substituted letters. BLAST evaluates the

* This project was supported in part by the National Science Foundation under NSF Grant EIA-9985986. The opinions expressed are those of the authors and not necessarily those of the foundation.

† Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1 Throughout this report we assume the alphabet is A, T, G, and C representing the nucleotides adenine, thymine, guanine, and cytosine. BLAST operates on other alphabets as well, such as the 22 amino acids found in proteins. All of the alphabets supported by BLAST are implemented from the same code base, so there is no loss of generality due to our assumption.

```
          Query Sequence
      AGCTTTTCATTCTTGACTGCAACG

         Subject Database
...AGCTTTTCATTCTGACTGCAACGGGATGTC...
```

(a)

```
Score = 32.2 bits (16), Expect = 0.005
Identities = 23/24 (95%), Gaps = 1/24 (4%)
Strand = Plus / Plus
Query: 1  agcttttcattcttgactgcaacg 24
          ||||||||||| ||||||||||
Sbjct: 1  agcttttcattc-tgactgcaacg 23
```

(b)

Figure 1: (a) sample input and (b) BLAST output

statistical significance of the variances and reports all of the high-scoring matches found in the subject. For example, consider the query and (a portion of) a subject database shown in Figure 1(a). BLAST recognizes the similarity and its output is reproduced in Figure 1(b).

BLAST is not the fastest tool. Nor is it the most accurate. Nonetheless it remains exceedingly popular with biologists. This is in part due to the fact that it was one of the first tools and that the statistical integrity of the heuristic was quickly established. This led many scientists to learn the heuristic as an abstract tool. That is, one can adjust the parameters, read the output statistics, and immediately interpret the results without considering the details of the heuristic. In addition to simply being familiar this makes scientists more productive. The popularity — and efficiency afforded by abstraction — was clearly fueled by the fact that the source code to BLAST was open and available. This enabled large numbers of scientists to download the code and run it on their general-purpose hardware. It became a mainstay of the field because it could be compiled on the most powerful (and most expensive) machines as well as the least expensive desktops. Pragmatically, this feat would not have been possible if the source was unavailable.

This, however, is not the complete picture. Indeed, most medium to large bioinformatics laboratories also include special-purpose computing machines to do similarity searches [15]. As one would expect, a special-purpose machine is much faster than BLAST running on general-purpose computer. The labs use these type of machines when turn-around time is important or when they want to manually prioritize their job mix. These machines, almost universally, use FPGA devices to implement their (closed source) algorithms. FPGAs are programmable logic devices that can be used to implement custom-hardware computing machines. They are commodity parts used in a wide range of products from digital cameras to high-speed Internet routers to the rovers recently landed on Mars. They are general-purpose in the sense that they can be reprogrammed repeatedly (every application run) and quickly (on the order of milliseconds). Yet these devices remain flexible enough to realize custom hardware to accelerate an application. Several vendors market standard PCI bus cards with user-programmable FPGAs for this purpose. Using FPGAs to build custom computing machines is commonly referred to as Reconfigurable Computing (RC).

This brings us to the central question of this work. If the open source availability of BLAST has been instrumental to its success and FPGA-based systems are known to accelerate similarity searches, what technical barrier, if any, prevents an open source FPGA implementation of the BLAST heuristic? More specifically, this report investigates the portability and cost-effectiveness of an FPGA-based implementation BLAST. Implicit in the cost-effective term is the price and performance of the implementation. A portable implementation is one where the non-recurring engineering costs associated with moving the implementation from one FPGA board to another is limited to interfacing issues and do not involve reworking the design. The primary contribution of this work is establishing the feasibility of the this approach but, almost as important and valuable, is an artifact of our study: an open source distribution of the implementation the design. The design and associated documentation can be found at: `http://www.ittc.ku.edu/rcblast`

The rest of the paper is organized as follows. The next section provides a brief introduction to Reconfigurable Computing and FPGAs. In section 3, the run-time behavior of BLAST is analyzed and a generic hardware design is presented. In section 4, an experimental setup is described and the resulting performance reported. Based on the analysis we present our conclusions in section 6, along with a brief description of future work.

2. Reconfigurable Computing and FPGAs

Reconfigurable Computing (RC) employs FPGAs to build a custom computing architecture. Field Programmable Gate Arrays (FPGAs) are integrated circuits consisting of an array of configurable logic blocks (CLBs) attached by a programmable interconnect (as shown in Figure 2). The programmable interconnect uses wire segments and switch boxes to propagate signals between logic blocks. Digital circuits are mapped to the CLBs which consist of look-up tables (LUTs) and flip-flops (FFs). These can be configured to implement the arbitrary combinational logical and state machine circuits. In addition to LUTs and FFs there are a number of dedicated (fixed) hardware circuits to handle specific functionality, such as fast ripple-carry adders. The Xilinx 4085XLA (used in the experiments presented here) is

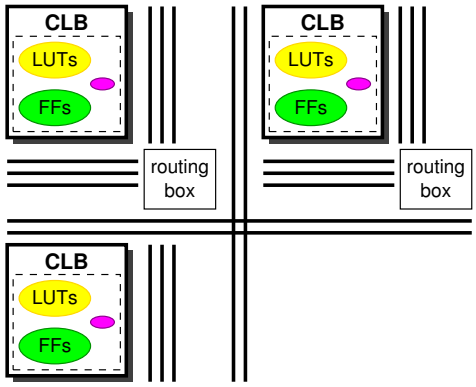

Figure 2: abstract view of an FPGA

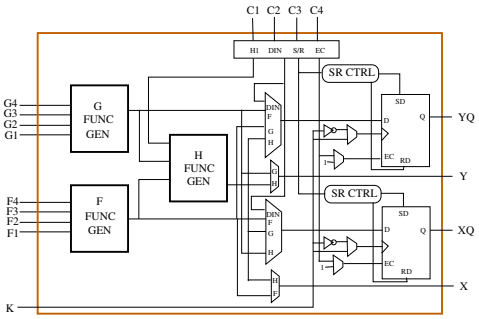

Figure 3: A Xilinx 4085XLA Configurable Logic Block (CLB)

a 85K logic gate device with 3136 CLBs in a 56×56 array. Its detailed CLB structure is shown in Figure 3.

It is possible to configure these devices by specifying the individual LUTs and configuration lines but for larger designs, such as the BLAST accelerator, a Hardware Description Language (HDL) such as Verilog or VHDL is a practical requirement. A suite of vendor tools is used to convert an HDL specification into a bit stream suitable for programming an FPGA. Developing hardware requires special training and this presents a significant barrier to software programmers. However, once a specific application (such as BLAST) is created the source can be widely distributed. By analogy, one might observe that not every biologist could write the original BLAST source code but many can compile and use it.

The cost of using programmable logic (instead of directly implementing the digital circuit in an application-specific integrated circuit) is an increased number of transistors and slower signal propagation — both due to the cost of introducing configurability. Hence, a central theme in RC is overcoming these disadvantages with architectural performance gains. Performance gains typically come from specialized circuits, pipeline parallelism, and spatial parallelism.

3. Analysis and Design of RC-BLAST

While it is possible, it is usually not feasible (nor beneficial) to implement the an entire application in the FPGA. More common is the practice of identifying the computationally intensive part of the application by analyzing the code or profiling the application at run-time. Then, these critical segments are targeted for acceleration with the FPGA resources. In this section, the run-time behavior of BLAST is analyzed and based on these results a generic hardware design is proposed.

3.1. Run-Time Analysis

BLAST comes in various 'flavors' that handle different biological databases (nucleotides, proteins, etc.). These flavors of BLAST are all part of a single software package available from NCBI[9] and share a common code base. The NCBI 2.2.6 version of BLAST, which is used here, consists of approximately 1545 C source files in 210 directories and a complex set of scripts to compile the application on a variety of platforms. To analyze the run-time of behavior of BLAST, we compiled the source code with gprof, a sample-based profiler, enabled. (All of the profiling experiments were conducted on the machine eventually used for the implementation; see the next section for details.)

Along with numerous optional arguments, BLAST requires three command-line arguments including the program name, database file and query file. The program name determines what flavor of BLAST the user wants. For example **blastp** compares a peptide query sequence against a database of protein sequences. To expedite our investigation we looked at the simplest one, **blastn** which compares a nucleotide query sequence against a database of nucleotide sequences. Using a test query sequence that comes with the BLAST source code and a small (1.3 MB) database ecoli.nt.2 downloaded from NCBI[9], it was found that one subroutine (**BlastNtWordFinder** in the file **blast.c**) contributed about 80% of the total execution time. From this starting point, we considered this subroutine to be the critical segment. Previously, colleagues in our lab had looked at ways of increasing this critical segment so that it would constitute a larger percentage of the execution time. By incorporating the subroutines that come after **BlastNtWordFinder**, they were able to increase it to 97.2% of the execution time (for some sample input) which gives them a theoretical upper bound of $35\times$ performance gain. This was crucial for their project because they had to justify much more expensive hardware.

Table 1: Size of Sequences

Subject Seq	Size
month.nt	284 MB
drosoph.nt	121 MB
ecoli.nt	1.3 MB

Query Seq	Size
RNase A (NM_002935.2)	716 Letters
Random Query	1300 Letters
FSHR (NM_181446.1)	2196 Letters

Table 2: percent execution time of `critical_code` for nine queries

	RNase A	FSHR	Random
drosoph.nt	72.20%	51.62%	75.12%
month.nt	69.15%	46.24%	75.72%
ecoli.nt	73.00%		

In contrast, our target is almost two orders of magnitude less expensive. Hence, we took the opposite course. Instead of growing the critical segment, we isolated about 120 lines of **BlastNtWordFinder**. These 120 lines were subsequently moved into a separate function named **critical_code**. By repeating the profiling experiments, we were able to establish that **critical_code** still effectively accounted for 80% of the total execution time. Further analysis led us to isolate 26 lines of the original **BlastNtWordFinder** function that represents about 73% of the total execution time for our sample query and subject database. This gives us a theoretical maximum speedup of $3.7\times$ which would be sufficient for our cost-effectiveness argument. Furthermore, the 26-line version of **critical_code** results in a simpler design that significantly improves the portability.

To verify that the importance of **critical_code** was not simply due to our choice of sample data, we conducted an array of experiments involving three queries and three subject databases. The three subject databases and their sizes are shown in Table 1 (all from NCBI). Two of the three queries also came from NCBI. The Random Query was a synthetically generated from a random sequence of A, T, G, and C letters. The query sizes are also listed in Table 1.

We tested every query against every database resulting in nine profile data points. In every case **critical_code** was the most important subroutine and for a majority of the cases, it was over 73% of the execution time. Table 2 summarizes the data obtained by profiling the BLAST source code. Complete data is available[8]. It is observed that the total execution time spent in this function increases with the size of the subject database. This suggests that larger databases will benefit more from this approach.

3.2. Generic FPGA Design

Now that we have one **critical_code** function which is the computationally intensive segment of the complete BLAST source code, we need to implement this function as a circuit for the FPGA. Using VHDL we designed an equivalent digital circuit which is described below.

Each database file or subject sequence is made up of many subsequence sections. The BLAST heuristic works by constructing a lookup table from the query. In **BlastNtWordFinder** routine the BLAST algorithm loops over all the sections in the database file and compares each one of them with the query sequence.

The lookup table (also called the hit table) is built from the input query sequence. Each row of the lookup table contains a word, made of eight consecutive characters from the query sequence. Words of 16 bits (8 letters) are formed from the subject sequence by traversing it in four letter hops. Each word is then used as an index into the (64K entry) lookup table. The table is encoded to indicate every place where that word appears in the query. If the count of occurrences of the word in the query sequence is zero, the current word does not occur in the query and is discarded. If the count is non zero then each offset is retrieved from the table. For each offset in the query sequence the subject and query words are extended to the left and right. If the comparison routine generates a high score, then the the subject and the query words, as well as their offsets, are passed to another routine called the BlastNtWordExtend. The task of indexing the newly created words from the subject sequence into the lookup table and retrieving the query offsets of these words is implemented in the new **critical_code** function. The block level view of the implementation in the FPGA is shown in Figure 4

In the present hardware design implemented the BLAST application is first started by typing the **blastall** command on the host processor. This will build the lookup table using the query sequence and loads the lookup table onto the SRAM in the ACEIIcard. Next as the execution reaches the **critical_code** function one subsection of the subject sequence is transferred to the design on the FPGA, where in it identifies all the basic eight letter hits between this subsection and the query sequence. After transfer the host processor just sits idle till these hits information is obtained from the FPGA. The remaining extension of hits operation is performed on the host processor and the results are written into the output results file.

Now we explain the hardware design made using the hardware description language, VHDL. The complete hardware design consists of two state machines one on the input side *addr_smc* and other *hitinfo_smc* on the output side. The word inputs to the design are stored in the top half of a 64 bit register named *subject_buffer* as shown in the Fig-

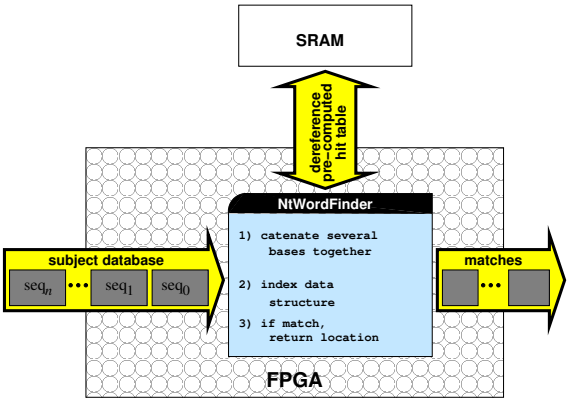

Figure 4: WordFinder

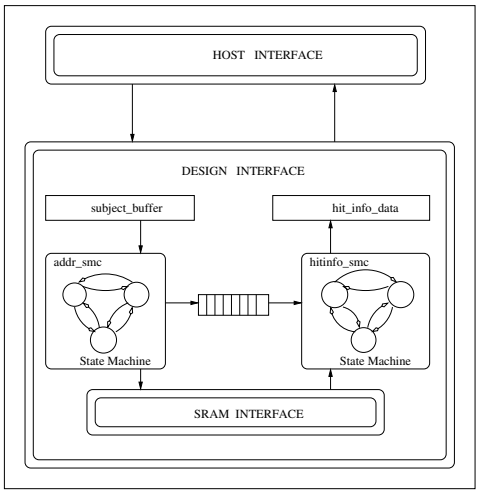

Figure 5: Design in Hardware

ure 5 and gradually shifted to the right in steps of four characters. The *addr_smc* reads four characters at a time from the lower end of this 64 bit input register and generates addresses to index into the lookup table in the SRAM. This state machine also keeps track of the subject offset of these words by writing them into a FIFO.

The *hitinfo_smc* state machine reads the count value and the query offsets returned from the SRAM and outputs them to the software routine. This state machine also pops off the subject offsets of these words from FIFO.

3.3. Interface

The lookup table built from the query sequence is loaded into one bank of SRAM available on the ACEIIcard. The bytes of SRAM contain the lookup table as shown in the

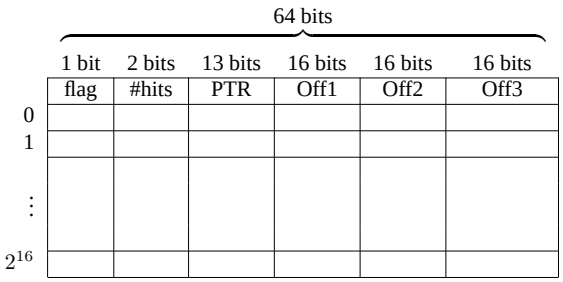

Figure 6: Lookup table in Hardware in Left SRAM

Figure 6. Information from each row of the lookup table is stored in 8 bytes or 64 bits of SRAM. The first three bits contain a single bit flag and a two bit counter of the number of occurrences of the 8 character word of this row of the lookup table in the query sequence. Bits from 16 to 63 contain the three offsets in the query sequence. Thus if the count is more than 3, the flag bit is set and the next 13 bits contains a pointer to the location where the additional offsets are stored. As of now the hardware design assumes that no word in query sequence is repeated more than three times, and the design stops running with an error message if the query sequence has a word repeating more than thrice.

After making all this design, the last step is to remove the **critical_code** function from the BLAST source code and replace it with a hardware interface stub that (1) copies the lookup table into the SRAM on the target board (2) streams in the subject sequence as input to the design in the FPGA and (3) reads back the hits information from the board.

4. Evaluation

To make an accurate evaluation of the portability and cost-effectiveness of the proposed design, it is necessary to realize the design in a real system. Below we describe the AceIIcard implementation followed by a discussion of its portability and cost-effectiveness.

4.1. Implementation

The generic design presented in the previous chapter was implemented. An Intel architecture based, i386 machine is used for both comparison and as a host for the FPGA-based solution. The machine runs on Red9.0 Linux operating system. The hardware platform used for implementing the design is called Adaptable Computing Engine or just the AceIIcard.

The AceIIcard is a PCI bus-based card with two Xilinx XC4085XLA FPGAs and two blocks of SRAM each of size 1MB. All the components on the card are connected by a local I960 bus that is bridged to the host's PCI bus. This architecture is illustrated in Figure 7. For our tests, only

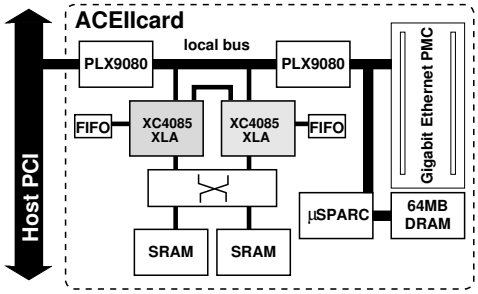

Figure 7: AceIIcard

the left FPGA and SRAM bank are used. On the host machine, an AceIIcard device driver is loaded. To make sure that the time measurements are accurate, all the BLAST experiments are run with no other user programs running.

To verify the correct behavior, an unmodified copy of BLAST, downloaded form the NCBI site, was compiled and run on the host machine. Designated SW-BLAST in this section, the output was compared against our RC-BLAST implementation. Given the same input, the two versions produced the same output.

4.2. Portability

It is difficult to quantify how portable a design or a piece of software is without implementing it on many instances. Without access to many hardware platforms, this has to be a mainly a theoretical discussion. By portable, we mean that the non-recurring engineering costs associated with moving the implementation from one FPGA card to another is just limited to interfacing issues and do not involve reworking with the design. As mentioned in subsection 3.1, previously colleagues had worked on larger BLAST designs the explicitly spanned multiple FPGA chips. The first version, designed for an Annapolis Micro Systems Wildforce card, used three chips. The second version used both chips in the AceIIcard. In both cases the design necessarily had to incorporate board-level architectural features, making them less portable. In contrast, our design — which is significantly smaller — entirely fits on a single chip. The design does not use any special-purpose components, making it relatively easy to port to other device families. The only remaining board-level features that it uses is the RAM interface but almost all boards include that as part of the general host interface. Thus we do not anticipate any special challenges in porting this design to any number of typical cards available today.

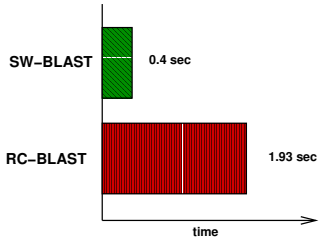

Figure 8: RC-BLAST v. SW-BLAST performance on a small query

4.3. Cost-Effectiveness

Cost-Effectiveness is a measure of the price of computing power. It is often used to compare two options where the question is either (i) Given a fixed budget, which option provides more computing power? or (ii) What is the least expensive way of achieving some performance objective? Both questions can be answered by considering the cost/performance ratio.

The execution time for one run of BLAST (using the small query and small database) is shown in Figure 8. The results show the RC-BLAST system running significantly slower that the software approach. Since the FPGA board cost $6,000 (new), clearly it is not possible to argue that the Ace2card is a cost-effective solution. However, this implementation still gives us the opportunity to to investigate the source(s) of the problem. We note several observations.

Memory Speed. We note that since colleagues began working on the project several years ago, processors have improved dramatically. With the 512KB of L2 cache on the Pentium III and modern clock speeds, the processor can perform a look-up about as fast as our FPGA hardware. The slowest part of the current design is the external RAM access to the lookup table. However, the BLAST hardware could be designed so that the vast majority of the lookups do not need to go to external RAM. The results indicate this additional effort would be useful.

Disk Speed. Upon investigation, we realized the PCI bus presents a significant bottleneck. When the project began, the CPU and system memory dictated BLAST performance. It now appears that storage is the slowest component — both systems RC-BLAST and SW-BLAST would run faster if secondary storage was faster. Since realistic databases cannot be cached in primary storage, this must be included. (Note that profiling does not account for time when application is blocked on I/O.) Our AceIIcard is further disadvantaged because the disk and the FPGA are on the same peripheral bus. This means that the database traverses the bus multiple times (from disk to primary memory and processor and then again on its way to the FPGA). Separate

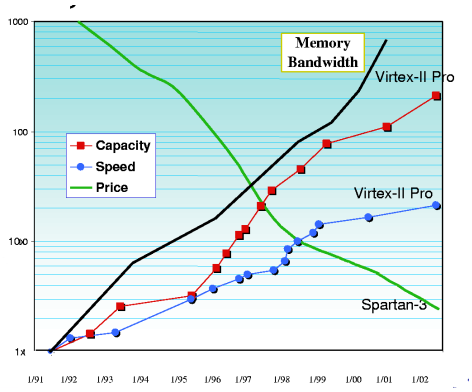

Figure 9: basic gate cost (proportional to 4085 CLB) for FPGA technology

buses is one solution but even better is to add direct disk access to the design. Unfortunately we are unaware of any current host interface designs that support this and while some have suggested it, the resulting design may end up being vendor-specific.

Resource Utilization. One of the goals of our design was to make it simple, small, and portable. However, in hindsight, our design dramatically under utilizes the CLB resources of the device. Of the 3136 CLBs, the host interfaces consumes 621 of them while the match unit only uses 83. This means that over three-quarters of the chip is unused and newer FPGA devices are considerably larger. Hence, we suspect there are several ways this design could be modified to exploit the additional resources.

While the current results are clearly discouraging, it is important to consider the trend. The first card used in the lab cost approximately $35,000. The next card, the AceIIcard used in these experiments, was purchased in 1999 for approximate $6,000. Several cards available today are less than $500. In general the cost per CLB has dropped dramatically — much faster than other computer technology (see Figure 9).

5. Related Work

A complete description of database search algorithms is beyond the scope of this report however it is important to note that other algorithms and projects have similar aims.

The Smith-Waterman (SW) algorithm finds the most similar subsequences of two sequences (the local alignment) by dynamic programming [7]. However these optimal computations requires execution time in the order of quadratic time [14].The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another.

The FASTA algorithm is an approximate heuristic algorithm used to compute suboptimal pairwise similarity comparisons. Dynamic programming is used to compute a series of subsequence alignments called hotspots which are combined to approximate a larger sequence alignment and global similarity score. Although not as optimal as the Smith-Waterman algorithm, the FASTA algorithm nevertheless executes in more rapid time and thus offers a trade off between comparison accuracy versus execution time [5, 6, 6, 12].

Since the introduction of BLAST, it has become very popular with the exception of FASTA, it is probably uncontested in practice. BLAST is computationally intensive, it runs slowly. A research publication [7] from Nanyang Technological university claims that the dynamic programming used in BLAST can be mapped efficiently to a linear array of processing element. TurboBLAST, describes a parallel implementation of BLAST suitable for execution on networked clusters of heterogeneous PCs, workstations or Macintosh computers [3]. The design of BLAST++ by National University of Singapore is based on the observation that the seed searching step of BLAST is a bottleneck that consumes more than 80% of the total response time [16]. Previous work on this project helped in identifying the software routines in BLAST that take most of the computational time. These routines if implemented in reconfigurable hardware will speed up the execution of BLAST significantly.

Researchers in reconfigurable computing were attracted to sequence matching for a number of reasons. Perhaps the most compelling was that the sequence matching problem was computationally intensive and had no floating point operations. Earlier some research institutions have explored ways to implement the sequence matching problem on FPGA hardware. A computing and modeling unit in Rome designed a special purpose processor for PROtien SImilarity DIScovery, called PROSIDIS using FPGAs [2]. This design on FPGAs can be used as HW booster for protein analysis algorithms as its operations are not efficiently supported by conventional processors. A research group from a German University claims that in contrast to pure software approach, the parallel architecture using a PCI based hardware accelerator gains a speedup of factor of 500 in detecting an optimized set of primers for a given gene sequence [13]. These primers are used in making DNA Chips.

6. Conclusion

The long-term goal of this work is to determine the feasibility of making an Open Source FPGA-based accelerator for BLAST. Several design options were considered and the run-time behavior of BLAST was determined by profiling the executable. In this contrast to previous work, this paper proposed a simple hardware design involving a small

but computationally significant portion of the BLAST code. This design was implemented on a PCI bus based FPGA card and interfaced the the the BLAST application. The cost-effectiveness of this implementation and the portability of the design were evaluated. While the simple design appears to be very portable, the implementation was not cost-effective. However the implementation led to several observations that will help guide the next design. The first is that the presence of large on-chip caches will give modern processors a competitive edge over FPGAs that use external memory. Current FPGAs also have on-chip RAM but the proposed design does not take advantage of this resource. The second observation is that BLAST has become an I/O-bound problem. Profiling the code did not reveal that the software-only version of BLAST could be sped up with a faster disk subsystem. The current system works by reading the subject database from disks attached to the PCI bus and then sends the data back over the PCI bus to the FPGA card. This approach is not viable since the bandwidth of PCI bus significantly limits the performance. The third observation is is that while the simple design is portable, it significantly under utilizes the gates available on the FPGA. Since the cost per gate is dramatically decreasing, this is a resource that could significantly impact the FPGA's cost-effectiveness measure. From these observations, we draw several conclusions about how to enhance the current design. Clearly, the lookup table data structure needs to take advantage of the on-chip FPGA RAM to remain competitive with on-chip caches. FPGA designs that directly control (multiple) Serial ATA disks drives are emerging and this presents an excellent opportunity to improve the FPGA's cost-effectiveness. Finally, the BLAST heuristic inherently has a significant amount of parallelism. Assuming the bandwidth into the FPGA can be increased then the large unused gate resources of the FPGA could be leveraged to increase performance through parallelism. In summary, the AceIIcard does not provide a cost-effective solution but the implementation and current technology trends suggest that the long-term goals are feasible.

Acknowledgements

Most of the RC-BLAST work was completed when all of the authors were associated with the PARL lab (`http://www.parl.clemson.edu/`) at Clemson University. We wish to thank several people that contributed to this project. Andrew Mehler did the first profiling work and contributed to the Wildforce board design. Katrina Logue translated that design into the two-chip AceIIcard solution. Brian Greskamp improved the Linux device driver. Several colleagues in the lab helped with specific aspects of the design and in debugging the hardware. Support for the FPGA hardware in the lab came from an NSF Instrumentation Grant EIA-9985986 and donations from Xilinx, Inc.

References

[1] S. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman. A basic local alignment search tool. In *J.Mol.Biol.215,3,403-310.*, 1990.

[2] P. A.Marongiu and V.Rosato. Prosidis: a special purpose processor for protein similarity discovery. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.

[3] R. D. Bjornson, A.H.Sherman, S.B.Weston, N. Willard, and J.Wing. Turboblast : A parallel implementation of blast built on the turbohub. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.

[4] U. Food and D. Administration. *Parkinson's Disease New Treatments Slow Onslaught of Symptoms*. url:http://www.fda.gov/fdac/features/1998/498_pd.html.

[5] D. Gusfield. *Algorithms on strings, Trees and Sequences*. 1997.

[6] D. J. Lipman and W. R. Pearson. rapid and sensitive protein similarity searches. In *Science 227,4693,1435-1441*, 1985.

[7] B. S. Manfred Schimmler and H. Schroder. Massively parallel solutions for molecular sequence analysis. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.

[8] K. Muriki. Design and implementation of open source fpga-based accelerator for blast. Master's thesis, Clemson University, Dec. 2004.

[9] NCBI. Blast software. Source may be downloaded from `http://www.ncbi.nih.gov/BLAST` and databases from `ftp://ftp.ncbi.nih.gov/blast/db/`.

[10] D. of Energy. *Medical Sciences Division Research Programs*. url:http://www.sc.doe.gov/production/ober/msd_res_topic.html.

[11] N. I. of Health. *Selected Research awards of NIH*. url:http://www.nih.gov/about/researchadvances.htm.

[12] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proc.Natl.Acad.Sci.USA 85,8,2444-2448.*, 1988.

[13] H. Simmler, H. Singpiel, and R. Manner. Real time primer design for dna chips. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.

[14] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. In *J.Mol.Biol. 147,1,195-197*, 1981.

[15] D. Time Logic. *Recent Product Citations*. url:http://www.timelogic.com/decypher_citations.html.

[16] H. Wang, T. H. Ong, B. C. Ooi, and K. L. Tan. Blast++: A tool for blasting queries in batches. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.