2009 17th IEEE Symposium on Field Programmable Custom Computing Machines

RC-BLASTn: Implementation and Evaluation of the BLASTn Scan Function*

Siddhartha Datta Parag Beeraka† Ron Sass
skdatta@uncc.edu beeraka@gmail.com rsass@uncc.edu
Reconfigurable Computing Systems Lab
University of North Carolina at Charlotte
9201 University City Blvd.
Charlotte, NC 28223–0001

Abstract

BLASTn is a tool universally used by biologists to identify similarities between nucleotide based biological genome sequences. This report describes an FPGA based hardware implementation designed to accelerate the BLASTn algorithm maintaining the same results yielded by the software developed at NCBI. A detailed profile study identifies the `Blast_Nt_Scan` *function as the computationally intensive part of the algorithm. A hardware component has been designed and implemented for this critical section. Rather then trying to implement more of the computation on the FPGA chip, our focus is on improving workload performance. Hence, the hardware has been designed to be replicated and placed on the FPGA to reduce initial comparison latencies between multiple short sequences (queries) and a subject database. Tests reveal the current implementation achieves an approximate* $4\times$ *speedup over the software run on a modern general purpose computer generating identical results.*

1 Introduction

The Basic Local Alignment Search Tool (BLAST) [1, 2] is a well-known bioinformatics application that is heavily used by biologists studying functional genomics and other related efforts. The Reconfigurable Computing field has a long history of using algorithms and heuristics from this application to motivate investigations. During 2005 [12] the first end-to-end results that exactly reproduce the results of the BLAST application were reported. Although the hardware component was slower than software, the results highlighted a common pit-fall. I/O bandwidth is an integral part of the problem. The application was close to being I/O bound and accelerating the computation simply exacerbated this problem.

Since 2005, technology has changed considerably. Bioinformatic databases have grown exponentially (see Figure 1), the computational capability of FPGAs have increased significantly, and storage (I/O and RAM) latencies are virtually unchanged. Most dramatically, major microprocessor manufacturers have warned that clock frequencies will not continue to increase as quickly as they have in the past twenty years. If this portends the death of frequency scaling, then the ramifications are enormous. One of the implicit tenets in high-end computing has been that the same (sequential) application (often the same executable) will run faster on future microprocessors in part due to faster clock rates. This portability has justified significant investments in software and presents significant barriers to alternative solutions. Interestingly, technology trends suggest that integrated circuits will continue scaling in other dimensions. In particular, the number of transistors will continue to increase. FPGA devices are well-suited to take advantage of the improvements in this area. Many tiled or array designs can easily be scaled with generic parameters that size the design to the available resources. An important corollary (from [12]) is that simply scaling the computational capability when larger devices becomes available may not improve the overall application. Achieving spatial scaling requires the original work to be constructed in such a way that the overall system performance will scale with adding resources. Often this cannot be captured in a set of simple generic parameters.

The primary focus of this paper is to improve the performance of the BLASTn by implementing the computationally intensive parts in the programming logic of the FPGA. Secondarilly, we are interested in exploring spatially scal-

*This project was supported in part by the National Science Foundation under NSF Grants CNS 06-52468 (EHS) and CNS 04-10790 (CRI). The opinions expressed are those of the authors and not necessarily those of the Foundation.

†Parag Beeraka is at AMD, Inc.

978-0-7695-3716-0/09 $25.00 © 2009 IEEE
DOI 10.1109/FCCM.2009.15

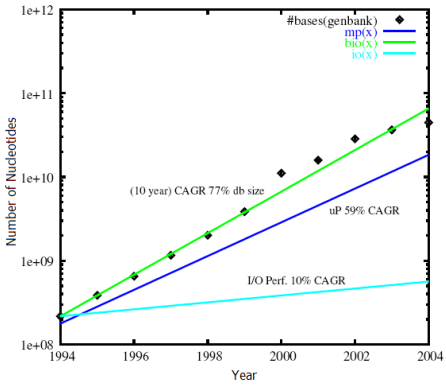

Figure 1. Growth in Database size Vs CPU and Memory growth

able designs. In section 2, details of the BLAST algorithm and previous research done in this area is discussed. Section 3 describes our design and its implementation. The results observed during the testing of our design is described in section 4 and conclusion in section 5.

2 Background And Related Work

The BLAST application was originally developed at Washington University, but in 1990, devlopment of the public domain code shifted to the National Center for Biotechnology Information (NCBI). The application, which is actually a collection of programs, are used to compare an unknown genomic sequence, coined as a *query* to an existing *subject* database to identify high similarity regions between them. The BLAST software has been developed in five flavors based on the nature of the subject database and query. These are: BLASTn, BLASTp, Blastx, TBLASTn, TBLASTx. BLASTn is the program used to make local searches between a nucleotide based query and subject database.

The architecture of the BLASTn algorithm is divided into three major stages. The first stage fetches a sequence of a subject database and identifies the regions of high similarity with a query. This stage passes the hit information comprising of the subject offset and query offset of the high similarity regions to the next stage. The second stage analyzes each region of high similarity and extends it bidirectionally to check for exact matches. This stage is called the ungapped extension stage. There are a set of regulations followed to allocate scores to the likewise pairs identified. The score alloted to a match on extension is positive. If there is a mismatch, a negative score is alloted. The calculated highest scoring pairs (HSP) are passed on to the next stage. The next stage runs the gapped extension. Here the extension is done and similarities are located with gaps in the alignment. The scoring scheme is the same with a penalty on the occurrence of a gap. There are a number of user defined parameters used to specify precision to the search. Significant parameters are the word size and stride length of each *W-mer* (word). The BLAST program outputs all the alignments found. The Estimate (*E-value*), number of extensions, number of successful extensions made and the total number of hits (similarities) found. An example of the BLAST output is shown in Figure 2.

```
>gb|AE000396.1|AE000396 Escherichia coli
K-12 MG1655 section 286 of 400 of the
          complete genome
           Length = 10098
 Score = 26.3 bits (13), Expect = 6.4
 Identities = 13/13 (100%)
 Strand = Plus / Plus
Query: 51   tgaagagcttaaa 63
            |||||||||||||
Sbjct: 4898 tgaagagcttaaa 4910
```

Figure 2. Output of the BLAST program

Biological Sequence searching has always been an active area of research. A lot of work has been done in implementing the Smith-Waterman [14] and Needleman-Wunsch [13] algorithm over hardware platforms to enhance the performance of search algorithms. The bottleneck for this algorithm resides in the first stage where the high similarity regions or hits are located. This issue has been addressed by Mercury BLAST [10], [4]. More work has also been done in implementing the entire application on the FPGA like Tree-BLAST [6, 5] and the TUC implementation [16, 15]. Another interesting design was proposed by[19] to port the entire application on chip. A commercial company, Time Logic [18] has designed the DeCypher machine which marketing literature suggests very impressive speedup numbers. Unfortunately they never released their design publicly. Recently, the IBM Cell broadband Engine[17] has also been used to speedup the BLAST program. A lot of work has been done in speeding up the BLASTp algorithm [8, 9, 7] using FPGAs. Due to its larger letter size, BLASTp is less I/O intensive.

All the mentioned implementations have show signficant speedups compared to software; however, most do so by compromising fidelity. That is, their results are not in complete agreement with the NCBI results. Mercury BLASTn[4] shows 98% to 99% accuracy in results, and Tree-BLAST[6] reports extra alignments (although reporting the same *E-values*). Another trade-off most FPGA implementations make is to restrict the query size. Extant implementations store the entire query on the FPGA which

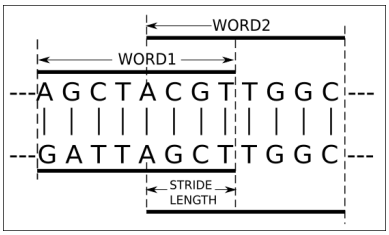

Figure 3. Demonstration of Word length and Stride Length

seriously limits the size of the query. While one could argue that BLAST is a heuristic and 100% compatibility with NCBI BLAST is unnecessary, it is difficult to convince biologists. Most users of BLAST have very little understanding of its algorithm. It is an opaque tool that has been thoroughly vetted. The typical user would have no idea whether the differences are statistically significant. Thus, in addition to just "better performance" there are still signficant challenges including complete compatibility with NCBI BLAST and arbitary sized queries.

3 Design And Implementation

3.1 Profiling and the Scan Function

The BLAST application has been profiled ([12]) in the past. However, in version 2.2.19 of the NCBI BLAST, the algorithm was recoded to be very efficient on RISC architectures. Hence, we profiled the new code. A set eight databases (ranging in size from 5MB to 4GB) each were searched with a set of five queries of different sizes. It was found that the `Blast_Nt_Scan()` function was the most computational-intensive part of BLAST. These tests were run on a dual core 2 GHz AMD Opteron processor running 64 bit Linux; depending on the test `Blast_Nt_Scan` consumed between 30% and 70% of the total execution time.

In BLASTn, only four letters are possible: A *(Adenine)*, C *(Cytosine)*, G *(Guanine)* and T *(Thymine)*. So the size of each letter is set to 2 bits. The primary function of the scan function is to stream the subject data sequence and locate hits. A hit is identified when the subject and query match at a particular location. Depending upon the the word length and stride length, lookups are made to an array where the hit information is stored. A word size is the number of letters which constitute to a group of letters based on which, comparisons are made. The stride length is the number of letters between the words where the comparison begins. Figure 3 shows how eight letter words with 4 letter stride lengths are formed.

The lookups are made to a lookup table created in a previous function. The size of the lookup table is $[2^{wordsize} \times wordsize]$. The query is always mapped to a fixed sized hash table irrespective of the query size. The query is formatted and compressed to hexadecimal and the offset of each query word is loaded in the table at the position number as the query word. The rest of the table is loaded with a default value fixed to -1 (0XFFFF). The hit information is the subject offset position and the query offset position for a particular subject data word. The scan function is written to accommodate various word sizes and stride length. The word length is set to eight. The software implementation is a `for` loop traversing the entire database sequence length making eight lookups, checking for hits and incrementing the subject offset by 32 for every cycle of the loop. The timing analysis of this function run on a state of the art 3.2GHz ($0.3ns$ clock time) Intel Pentium Processor with 1GB RAM and 512KB L1 cache(assuming 100% cache hit) is as illustrated below.

Let T_{Total_SW} be the minimum number of clock cycles to compare eight bytes of the subject data, T_{Loop_Comp} be the minimum number of clock cycles to execute the jump statement in the loop, $T_{Subject\_Offset}$ be the minimum number of clock cycles to update the subject offset, T_{Lookup} be the minimum number of clock cycles to make a lookup for a hit to main memory and T_{Hit_Check} be the minimum number of clock cycles to check for a hit. The lookup is made to main memory and may consume more then one clock cycle, but if in the cache, may consume only a clock cycle. Hence, we assume the lookup time is one clock cycle. Hence,

$$T_{TotalSW} = T_{Loop\_Comp} + 8 \times [T_{Lookup} + T_{Hit\_Check}] + T_{Subject\_Offset} \quad (1)$$

Therefore,

$$min(T_{TotalSW}) \approx 18\text{clock cycles} \approx 5\text{-}6\text{ns} \quad (2)$$

for comparing eight bytes of the subject data.

3.2 Single Core Design

3.2.1 Simple Core

The complete setup of the FPGA system is shown in Figure 4. The word length is fixed to eight letters (16 bits) with a stride length of four letters (eight bits). Designs like [10, 4, 6, 5] fix the word length to eleven. There is a greater level of precision in the final results generated with the word length set to eight as compared to eleven. The major components of the hardware core are an input FIFO, output FIFO and the main hit controller. The input FIFO is used as a buffer to store the incoming database subject data continuously streamed into the core from the hard

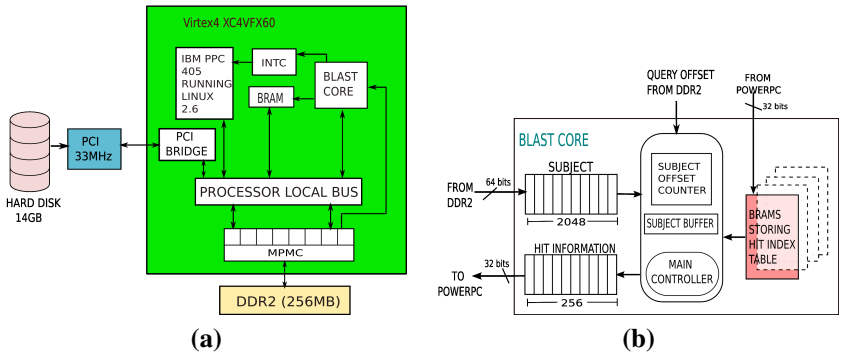

Figure 4. (a) top-level FPGA design (b) details of BLAST core

disk. A Direct Memory Access (DMA) is set up to fill the FIFO with eight bytes of data every clock cycle. If at any instance the FIFO is full, the DMA is stalled, till there is only one element left in it. This is implemented to ensure no overflow in the input FIFO. The output FIFO is used to store the hit information. The main hit controller is a finite state machine that coordinates all the functions of the hardware core. This state machine has four states. The first state pops a data element from the input FIFO onto the last eight bytes of the subject buffer register which is nine bytes wide. The second state makes a hit check for the current subject data element. There are eight lookups that need to be made per eight bytes of subject data (as explained in Figure 5). Lookups are made to the lookup table created in software and stored in the main memory. The size of this table is $2^{16} \times 16$ bits. Positive data fetched from the lookup table is the query offset (16 bits). This implies a hit. The main controller maintains a subject offset counter. This counter is incremented by four for every sixteen bits of subject data completing comparison. In the case of a hit, the lower sixteen bits of this subject offset counter is grouped with the lower sixteen bits of the query offset and stored in the outut FIFO. On completion of hit checks for all the eight bytes of subject data, the final byte is shifted left to the first position of the subject buffer register to group with next subject data loaded in. This ensures no lookups being missed. This is done in the third state of the FSM. The fourth state of the controller issues a read request to the input FIFO. This four state process repeats till the entire sequence of the database has been streamed. The hit information is read out by the PowerPC running he BLASTn program over the Processor Local Bus (PLB). The timing analysis of the core is as follows.

The core is clocked at 100MHz. Each clock cycle is $10ns$. Let T_{Total} be the total time time for the controller to pop $subjectdata_i$ to $subjectdata_{i+1}$. Let T_{Pop} be the time taken to pop one data element from the input FIFO

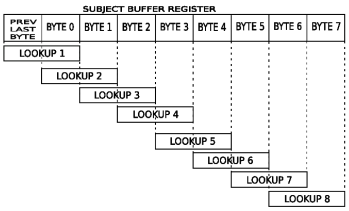

Figure 5. Demonstration of the lookups per subject data element

and load the subject buffer register, T_{Trip} be the time taken to fetch a lookup data from the main memory, T_{Lookup} be the time taken to make lookups for eight bytes of the subject data and Let T_{Shift} be the time taken to issue a read request to the input FIFO, shift the subject buffer register bits and update the subject offset counter. It is observed that T_{Trip} takes 24 clock cycles while T_{Pop}, T_{Lookup} and T_{Shift} take one clock cycle.

$$T_{Lookup} = 8 \times T_{trip} = 1920\text{ns} \tag{3}$$

Therefore,

$$T_{Total} = T_{Pop} + T_{Lookup} + T_{Shift} = 1950\text{ns} \tag{4}$$

This is extremely slow compared to the software run on the PowerPC. The next step in the design was to formulate a method to store the lookup table on the FPGA to avoid repetitive trips to main memory.

From equation 4, lookups to main memory for every eight bytes of the subject data takes about $2ms$. Typically a 2–4 GB nucleotide based database has about $200,000$ – $400,000$ sequences having lengths of around $1,000$ – $10,000$ bytes. This creates a maximum penalty of approximately 7800 seconds primarily due to trips made to main

memory. The first bit of each entry of this table indicates the presence of a hit. A Hit Index Table (HIT) is created by grouping the first bit of every entry of the lookup table into 32 bit words and storing them into a local Block RAM (BRAM) of size $[2048 \times 32]$bits on the FPGA via the PLB. This table is created during the first pass of the scan function in software. The overhead of loading this table into the FPGA is timed to be less than $1ms$ which is negligible compared to the total execution time of the program.

3.2.2 Cached Core

The second version of the hardware comparator stores the Hit Index Table in local BRAMs. The second state of the finite state machine is modified to calculate the address of the BRAM to check for a hit. The shift to the subject buffer register is done in the third state. The subject offset counter is incremented in the first state. The timing analysis of this design is as follows:

Let T_{Addr_Calc} be the time taken to calculate the BRAM address and T_{Hit_Check} be the time taken to make a check for a hit to the Hit Index Table. It is observed that T_{Addr_Calc} and T_{Hit_Check} take one clock cycle.

$$T_{Total} = T_{Pop} + 8 \times [T_{Addr\_Calc} + T_{Hit\_Check}] = 170\text{ns} \quad (5)$$

From equation 4 and equation 5, the performance of this design is more than 10 times faster than the non cached non pipelined design at the expense of increasing the BRAM block count of the core by 2% on the Virtex4-XC4VFX60 FPGA. It is about 25 times faster than the PowerPC running at 300MHz but about 30 times slower than the Intel Pentium.

3.2.3 Cached and Pipelined Core

There was a requirement to further speedup the hardware comparator's performance. Four dual ported BRAMs are loaded with the hit index table. The finite state machine makes eight lookups in parallel. In the case of a hit, the subject offset counter value corresponding to the subject data is stored into a buffer and the query offset data is fetched from the main memory. An interrupt goes off each time data is written to the output FIFO. Based on this interrupt the PowerPC issues a read command from the output FIFO. Figure 4 shows the final design of the hardware core. The timing analysis of this design is as follows:

$$T_{Total} = T_{Pop} + T_{Addr\_Calc} + T_{Hit\_Check} = 30\text{ns} \quad (6)$$

The main controller is three staged pipelined to accomodate rapid handling of the subject data. The initial delay

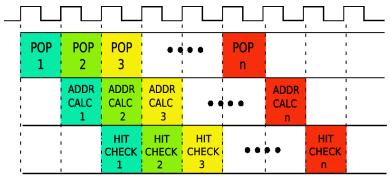

Figure 6. Pipelined comparator of the core

of the pipeline is two clock cycles, i.e. $20ns$. The first stage of the pipeline pops the first subject data element. The second stage calculates the BRAM address in order to make a hit check and pops the second subject data element. This BRAM address is stored in a register to be used for a hit check in the next clock cycle. In the third stage of the pipeline, the hit check is done based on the BRAM addresses calculated in the previous stage, BRAM addresses are calculated for the current subject data and the next subject data is popped. Figure 6 demonstrates the working of the pipelined core for every clock cycle.

So, the total time for the entire sequence of length N bytes is:

$$T_{Total\_Time} = 20\text{ns} + [10 \times (N/8)]\text{ns} \quad (7)$$

This results in an average speed of $\approx 1.25ns$ for comparison of every eight bytes of subject data. Comparing equations 2 and 7 the cached pipelined design of the hardware comparator proves to be $5 - 6$ times faster than the Intel Pentium. This is verified in table 1 where the scan function has been timed for different systems.

3.3 Multiple Cores

The subject database is transferred to the hardware core at $12.8GB/s$. The core, clocked at 100MHz handles data at $6.4GB/s$. The size of the input FIFO is fixed to $[2048 \times 64]$ bits. This is large enough to buffer the subject data in the pipelined core, due to differing data rates at which the core handles the data and at which it recieves the data from the hard disk. The input FIFO utilizes 4% of all the total BRAM count available on a Virtex4-XC4VFX60 FPGA. The maximum number of hits observed for any sequence of a nucleotide based database is approximately 400 and sparsely placed. The hit information is read as soon as data is written to the output FIFO. The size of the output FIFO is fixed to $[256 \times 32]$ bits which utilizes 1% of the total BRAM count on the FPGA ensuring none of the hit information is overwritten.

The scan function ported to hardware provides speedup against a state of the art general purpose computer. The rest of the code is handled by the PowerPC. Amdahl's law[3] is used to make certain timing analysis of the algorithm:

Table 1. Time taken to process the Scan function for sequences of nt.00 (3GB) Database

Subject Length	PowerPC 405	HW core version3	Intel Pentium (3.2GHz)	Dual core AMD Opteron(2GHz)
3088 Bytes	1437μsec	5μsec	10μsec	8μsec
24383 Bytes	12306μsec	39μsec	77μsec	70μsec
31025 Bytes	15658μsec	53μsec	163μsec	152μsec
41359 Bytes	20874μsec	68μsec	224μsec	208μsec

The profiling results indicate a maximum 70% of the total execution time taken by the scan function. Let $I_{Program}$ be the total number of instructions executed by the program, I_{SW} and I_{HW} be the the number of instructions implemented in software and hardware respectively, $T_{Program}$ be the total time taken to execute BLASTn and T_{SW} and T_{HW} be the time taken in software and hardware respectively.

$$I_{Program} = I_{SW} + I_{HW} \tag{8}$$

$$T_{Program} = T_{SW} + T_{HW} \tag{9}$$

Ideally, $T_{HW} \approx 0$

$$T_{Program} = T_{SW} = 0.3 \times T_{SW} \tag{10}$$

The PowerPC running at 300MHz takes:

$$T_{Program} = I_{SW} \times 0.3 \times 3.33\text{ns} \tag{11}$$

$$= T_{Program} = (0.99 \times I_{Program})\text{ns} \tag{12}$$

The 3.2GHz Intel Pentium takes:

$$T_{Program} = (0.312 \times I_{Program})\text{ns} \tag{13}$$

Conclusions based form equations 11 and 13, the Intel Pentium proves to be 3–4 times faster than the current hardware implementation. Hence, reducing the time spent in the scan function solely does not improve the performance of the entire program running on the FPGA compared to other processors. Profiling results do not indicate any other functions utilizing in excess of 2% of the total execution time. The limitations on hardware resources restrict the implementation of the entire program on the FPGA.

The solution to this problem is solved by building tiled designs that scale spatially over the FPGA. The BLASTn program is used to compare multiple queries over multiple databases. When run on a sequential processor, the program has to be run multiple times. The advantage over these architectures offered by FPGAs is to create multiple instances of the hardware comparator and load them with different queries as shown in Figure 7.

The hit index tables are created for multiple queries and loaded into the BRAMs on chip during the first pass of the scan function. This creates overhead latency compared to the software but is negligible compared to the total execution time. The overhead was timed to be less than $2ms$ for eight $50KB$ $(61,683$ character strings) queries.

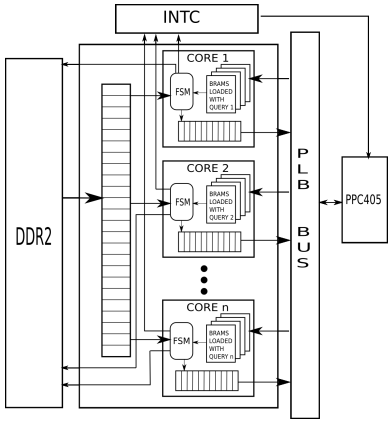

Figure 7. Design to increase throughput of the system

The database is streamed to the cores generating hit information for various queries and stored in the output FIFO of each core, during the first run of the program. The outputs read by the PowerPC are written to text files stored on the main memory. The output of the first core (the first query's hit information) is passed to the extension stage. For the remaining times the code is run, depending upon the queries, the respective text files are read bypassing the scan function completely.

4 Results

The implementation is done using Xilinx ml-410 boards. The Virtex4 XC4VFX60 FPGAs are used to port the designs. Xilinx Embedded development Kit (EDK) is used to develop and synthesize the designs. These devices have two RISC IBM PowerPC-405 32-bit processors fabricated on the die. Linux 2.6 is run on one PowerPC at 300MHz. The FPGA communicates with a hard disk through the PCI bus running at 33MHz. The hard disk has 14GB of memory and formatted using Ext2 file system. The resource utilization of systems with different versions of the hardware core and multiple hardware cores are given in Table 2. The maximum number of cores that can be packed onto the Vertex 4 is currently eight due to a limited number of BRAMs avail-

Table 2. Resource Utilization

Component	%BRAM Utilized (Core)	%Slice Utilized (Core)	%BRAM Utilized (System)	%Slice Utilized (System)
Hardware Comparator without Hit Index Table	2	4	31	45
Hardware Comparator 1 Hit Index Table	3	4	32	46
Hardware Comparator 4 Hit Index Table	10	5	43	48
System with 2 Hardware Comparator cores	18	7	30	49
System with 4 Hardware Comparator cores	34	12	61	54
System with 8 Hardware Comparator cores	65	23	92	66

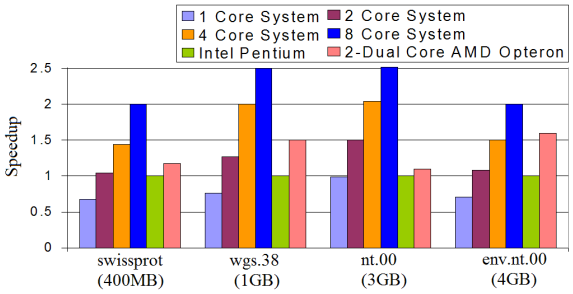

Figure 8. Speedups compared to the Intel Pentium running 16 queries of same size over several databases

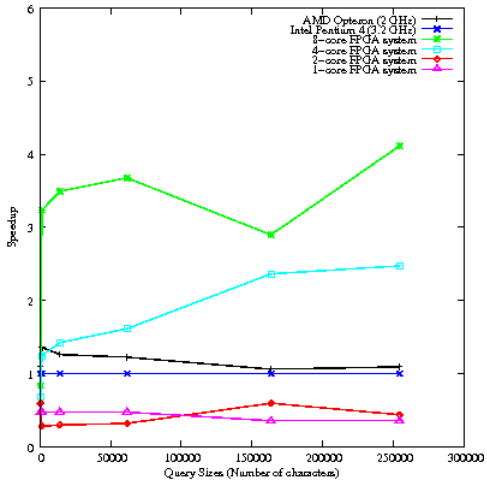

Figure 9. Speedups compared to the Intel Pentium running 16 queries of various sizes over month.nt (400MB) database

able.

A group of 16 queries, 500 Bytes (492 character strings) each are run against the *swissprot.nt* (400MB), *wgs.38.nt* (1GB), *nt.00* (3GB) and *env.nt.00* (4GB) databases downloaded from NCBI Genbank's website. BLASTn is run on systems with one, two, four and eight instances of the hardware comparator. These tests are compared to the performance of two other systems. The first system is the Intel Pentium processor discussed in section 3. The other machine is a 2-dual core AMD Opteron running at 2.0GHz with 512KB L1 cache and 6GB of main memory. The tests results are shown in Figure 8.

The graph is Figure 8 compares the speedup achieved by all the systems against the Intel Processor. The maximum speed up achieved is for 1-3 GB databases running on the system packed with 8 hardware cores. Proved in section 3, the single core system performs slower than the Intel and AMD. For the *nt.00* database, the system with a single core takes 2148 seconds, the dual core AMD processor takes 1797 seconds, the Intel processor takes 2114.4 seconds and the system with 8 cores takes 847 seconds. This is due to the system with 8 cores calling the scan function during two runs of the program as compared to the sequential processors running through the scan function during all 16 runs of the program. The system with 4 cores and 2 cores call the scan function during 4 and 8 runs of the program respectively.

The other test run, was with queries of various sizes run over the *month.nt* (800 MB) database. Each instance of the test involved 16 queries of the same size run over the database. With every instance of the test, the size of the queries were increased. The first instance of the test was with 16 700Byte (429 string characters) queries run against the database. This was followed by 16 10KB (1549 string characters) queries run against the database. The maximum query size considered was running 16 120KB (254,349 string characters) queries against the database. The speedup numbers compared to the Intel Processor recorded are shown in Figure 9.

Tests indicate the FPGA system with 8 cores' performance increases as the query size increases, due to cache misses that occur with the Intel and AMD processors. The Hit Index Table ensures a 100% hit. The tests imply that the performance of the system with eight cores depends upon three major factors, database length, query length and the

number of hits identified. As the query length increases, the FPGA system's performance rises compared to other sequential processors. This is due to their limited cache size and percentage of cache miss increase with the increase in query lengths. The number of hits identified affects the FPGA system's performance. If the number of hits is too large, more trips are made to main memorydropping the FPGA system's performance compared to other processors. The decline in Figure 9, for the 16 80KB queries test is due to this reason. This particular query when run against the *month.nt* database, resulted in $\approx$ 2.7 billion hits as compared to $\approx$ 350 million hits for the next query.

5 Conclusion and Future Work

The primary goal of this work was to accelerate BLAST workloads using an FPGA accelerator. A secondary goal was to investigate spatially scalable designs that will grow as FPGA resource increase (even if memory bandwidth does not). A memory-efficient design was presented that outperforms a modern day's general-purpose PC computer by a factor of $2-3\times$. While other FPGA/BLAST solutions exist, they impose restrictions, such as limited query sizes, or are not designed to scale. Moreover, most compromise fidelity. The proposed design is fundamentally scalable in both frequency and spatial dimensions. Furthermore, it is in complete agreement with the *de facto* standard NCBI BLAST. Future work includes expanding the parallelism in the database. Current work on I/O subsystems [11] are prised to unlock an enormous amount of bandwidth that will enable BLAST to run much faster.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–410, October 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, September 1997.

[3] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *Proc.Am.Federation of Information Processing Soc. Spring Joint Computer Conf*, pages 483–485, 1967.

[4] J. D. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. Chamberlain. Mercury blastn: Faster dna sequence comparison using a streaming hardware architecture. In *Reconfigurable Systems Summer Institute*, July 2007.

[5] M. C. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. V. Court. Single pass, blast-like, approximate string matching on fpgas. In *FCCM*, pages 217–226, 2006.

[6] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. Single pass streaming blast on fpgas. *Parallel Computing*, 33(10-11):741–756, 2007.

[7] A. Jacob, J. Lancaster, J. Buhler, and R. D. Chamberlain. Fpga-accelerated seed generation in mercury blastp. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 95–106, 2007.

[8] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury blastp: Accelerating protein sequence alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):1–44, 2008.

[9] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury blastp: Accelerating protein sequence alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):1–44, 2008.

[10] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the mercury system. *J. VLSI Signal Process. Syst.*, 49(1):101–121, 2007.

[11] A. A. Mendon and R. Sass. A Hardware Filesystem Implementation for High-Speed Secondary Storage. In *Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, Dec 2008.

[12] K. Muriki, K. D. Underwood, and R. Sass. Rc-blast: Towards a portable, cost-effective open source hardware implementation. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 7*, page 196.2, 2005.

[13] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1972.

[14] T. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[15] E. Sotiriades and A. Dollas. Design space exploration for the blast algorithm implementation. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 323–326, 2007.

[16] E. Sotiriades and A. Dollas. A general reconfigurable architecture for the blast algorithm. *J. VLSI Signal Process. Syst.*, 48(3):189–208, 2007.

[17] A. Wirawan, K. C. Keong, and B. Schmidt. Parallel dna sequence alignment on the cell broadband engine. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *PPAM*, volume 4967 of *Lecture Notes in Computer Science*, pages 1249–1256. Springer, 2007.

[18] www.timelogic.com. Time logic biocomputing solutions.

[19] F. Xia, Y. Dou, and J. Xu. Families of fpga-based accelerators for blast algorithm with multi-seeds detection and parallel extension. In M. Elloumi, J. Kng, M. Linial, R. F. Murphy, K. Schneider, and C. Toma, editors, *BIRD*, volume 13 of *Communications in Computer and Information Science*, pages 43–57. Springer, 2008.