Basic Local Alignment Search Tool for Nucleotides (BLASTn) for Hardware Acceleration

Alden Param
ajparam@cpp.edu

Jacob London
jelondon@cpp.edu

Mohamed El-Hadedy Aly
mealy@cpp.edu

Alex Chan
alexchan@cpp.edu

Simon Tutak
sdtutak@cpp.edu

Dr. Mostafa M. Hashim Ellabaan
mohel@biosustain.dtu.dk

Hmayak Apetyan
hapetyan@cpp.edu

Sivaramakrishnan Prabakar
sprabakar@cpp.edu

Electrical and Computer Engineering (ECE) Department
Cal Poly Pomona
Pomona, CA

Abstract*—The BLASTn (basic local alignment search tool - nucleotide) algorithm is widely used by researchers around the world. BLAST itself is used for a variety of purposes, but for our project we focused on BLASTn which targets nucleotide DNA sequences. In BLASTn the user inputs a sequence of DNA (the query), the program then compares this sequence to a database (the subject) of other sequences and returns the most similar DNA sequences. As DNA sequence and subject size increases, the time it takes to run the program increases exponentially. This is where our project comes in. Our project aims to accelerate the time it takes to run the BLASTn program. We first accomplished this in an implementation in Python, in order to understand BLASTn at a higher level. We then moved on to implementing the algorithm in a much faster language, C++. Throughout both of these implementations we came to learn that the bottleneck in the whole process was the execution of the Smith-Waterman algorithm. If we could accelerate that portion of the BLASTn algorithm, we would greatly increase the speed at which the program would run. That is why we decided to hardware accelerate the Smith-Waterman algorithm onto the Nexys 4 DDR FPGA using the Vivado design kit.

Keywords*—bioinformatics; BLAST; BLASTn; DNA; DUST; FPGA; genomics; hardware accelerator; nucleotide; Smith-Waterman;

I. INTRODUCTION

Since the completion of the first draft of the human genome in 2001, genomic data has been far outpacing Moore's law and is expected to reach the Exabyte scale with the next year and surpass even YouTube and Twitter by 2025 [1]. Not only is this data immensely valuable in the fields of personalized medicine, detection of genomic mutations that predispose humans to certain diseases, including cancer, autism, and aging, but also contributes to the understanding of molecular factors leading to phenotypic diversity (such as eye and skin color, etc.) in humans, and other life forms [1]. This growth in data is primarily due to DNA sequencing technologies. DNA sequencing refers to methods for determining the order of the nucleotide bases – adenine, guanine, cytosine, and thymine – in a molecule of DNA [31]. Sequencing is part of a broader science known as bioinformatics which is an interdisciplinary field that develops and improves upon methods for storing, retrieving, organizing and analyzing biological data using various algorithms [30]. In molecular biology, sequencing allows researchers to obtain information regarding gene mutations and disease and phenotype associations, among others [30]. In evolutionary biology, information obtained using sequencing can provide insights on how different organisms are related and how they evolved [31].

However, even with the advent of third generation sequencing technologies, it is difficult to process the overwhelming amount of genomic data. These technologies have prohibitively high computational costs; over 1,300 CPU hours are required to analyze the data using a reference, and over 15,600 CPU hours are required to assemble the reads de novo (or without reference) [1]. One of the most popular methodologies created to speed up this process is BLASTn, or Basic Local Alignment Search Tool for nucleotides. For DNA subjects and DNA queries, BLASTn is used. Even with processes such as BLASTn, it can take a significant amount of time to process all of the subject. Another issue with processes like BLASTn is that it can be inaccurate, as it is a heuristic algorithm and thus takes shortcuts to lower the processing time.

II. DEVELOPMENT

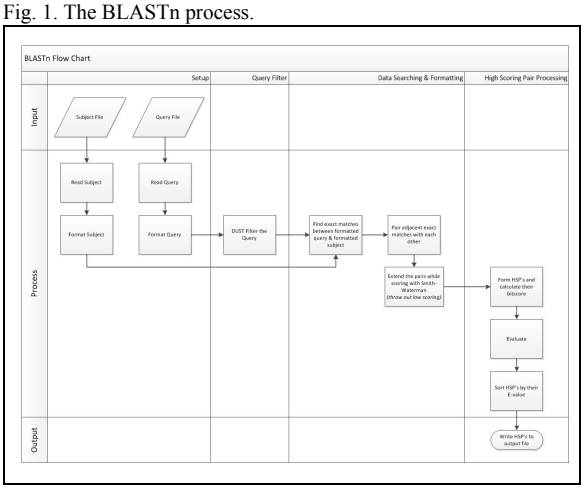
The development process went through four stages. The first stage was to understand the algorithm, working with the process on a white board and creating a flowchart to understand how different aspects of the algorithm worked, and what their purpose was. The second stage of the development process was to implement the algorithm in a high level language, in our case we chose Python. This was decided as errors were easier to spot and understand in a high level language compared to immediately attempting an implementation in a low level language or directly onto an FPGA. The third stage of the process was to implement the algorithm in a lower level language, for this we chose C++. The fourth and final stage of development was to identify the bottlenecks in the algorithm and attempt to hardware accelerate those portions.

III. ARCHITECTURE

The architecture portion gives a brief overview of the BLASTn process and then explains the components in detail later. First, we find all possible words in the query, score the words and then remove the low scoring words. Then the words go through DUST filtering to mask low complexity areas. After that, we find exact matches in the subject and pair nearby adjacent words that are exact matches and pass the nearness threshold (they are close enough to be valuable). We then extend the words to the entire query sequence and continuously score against the corresponding subject sequence while extending. If at any point, our score dips below the threshold we stop extending and throw away the pair. We then permute the query and score it with the subject counterpart sequences. Any low scoring matches are removed. We return the high-scoring pairs and calculate their E-values which indicate how relevant the scores are. The HSPs are sorted by increasing E value and are returned [3, 4, 15].

The BLASTn algorithm performs operations on DNA sequences in several steps, which together make up the parts of the algorithm. The process in which BLASTn operates is shown in Fig. 1.

Fig. 1. The BLASTn process.



A. Overall Architecture

Each of the steps of BLASTn are respectively as follows: sequence input, sequence splitting, DUST complexity filtering, sequence matching, sequence pairing, sequence extending, high-scoring pair formatting, sorting, and output. These sections were put together with the use of similar data-structures such as associative arrays and dynamic arrays. The definitions of a dynamic array can include a string, a dynamic array of characters, or any other collection of sequentially stored information. Each data structure used in this implementation is illustrated in Fig. 2 in the respective order of their definition below. A ‘:’ character refers to an association, an associative array is denoted by surrounding ‘{‘ and ‘}’ characters, and a dynamic array is denoted by surrounding ‘[‘ and ‘]’ characters.

In step one, sequence input, fasta or fa files are read from their files into an associative array where the name of each sequence is associated with its sequence data.

In step two, the sequence input is processed into an associative array where each sequence is

broken into words of length n. To decrease memory footprint, there are no duplicate words stored for each sequence. To solve this problem, each word is associated to the indices it appears at from its previous sequence.

In step three, the data from step two is processed with DUST filtering, then is processed with a matching algorithm. The DUST filtering algorithm [2] removes low complexity words from the previously processed query. After, this filtered query data is matched against the subject data with the following technique: if a query associative word, QAW, does not exist in any of the subject associative words, remove the QAW, else keep the QAW and move onto the next QAW.

In step four, the matches are processed according the the following technique: if the subject index of a QAW and its adjacent QAW's subject index are within the bounds of the query each word is in, then the two words can be considered to be a pair. This format allows for adjacent matches which are sufficiently close to each other to be kept. All other match pairs which are too far from each other are removed.

In step five, the adjacent pairs are extended together against their subject, and scored after each extension. This ensures that the resulting extended pair of matches are similar enough with the query they are in to be kept, otherwise the pair is removed.

Lastly, in step six, the fully extended pairs can be converted into high scoring pairs or HSPs. They are converted into high scoring pairs by recording data about each HSP such as their expected value or E-value and their bit score.

Finally, this data is sorted and written in standard Blast format to a user specified result file.

Fig. 2. Data Structure Format

1. $\{name : sequence\}$
2. $\{name : \{word : [indices]\}\}$
3. $\{subject : \{query : [Matched]\}\}$
4. $\{subject : \{query : [Paired]\}\}$
5. $\{subject : \{query : [Extended]\}\}$
6. $[HighScoringPairs]$

A. Smith-Waterman Algorithm

One of the most important parts of the BLASTn algorithm is an implementation of the Smith-Waterman algorithm. This is because the BLASTn algorithm is a technique that removes sequences which are unlikely to be related, before performing the Smith-Waterman algorithm. The time it would take to process a search against a large subject would be significantly less if the Smith-Waterman algorithm did not have to be performed on every single query. Once BLASTn has finished narrowing down which sequences are more likely to be a hit, the Smith-Waterman algorithm plays its part in the BLASTn algorithm to locally align the query to likely subject hits.

The Smith-Waterman algorithm takes two input sequences and performs a gapped alignment on them. To do this, there is a 2D matrix, where each axis length is the length of each axis plus one. This matrix is filled with zeros, and it will be traversed, starting at index (1, 1) so that row and column zero only contain zeros. Traversal starts, and each cell's score is determined by the greatest of the following: zero, sum of cell left and match score, sum of cell above and match score, or sum of diagonal-left and match score [24]. While this process is taking place, the location and value of the highest scoring value is recorded. When the matrix is filled, traceback occurs, where traversal starts at the location of the greatest value, and the corresponding movements, up, left, or diagonal, are performed until a zero is reached, where up or left corresponds to a gap, and diagonal corresponds to a match. At this point, the alignment has been finished, and the percent similarity and score for the alignment can be obtained [16].

This technique is very slow, but is quite accurate for the two given sequences. In order to speed the process up, there are a variety of techniques which many research teams [24, 25, 26, 27, 28, 29] have done to increase the speed of Smith-Waterman without losing any of its accuracy. The key technique is to parallelize the process by utilizing the new general purpose technology, which GPUs have to offer, which have shown a speed up of 37 times [29]. An additional technique is to utilize parallel processing on microprocessors, which have shown a speed up of six times to that of a single threaded version [23].

In the design presented in this paper, the Smith-Waterman implementation was created for use on an FPGA for a significant increase in speed and efficiency. This implementation is modeled after the design demonstrated in [32]. This would allow for the use of special purpose hardware for solving the problem of the Smith-Waterman algorithm's slow speed of calculation.

B. DUST Algorithm

DUST filtering is a module of the BLASTn algorithm that is used to mask low complexity

sequences. Low complexity sequences might result in high Smith-Waterman score outputs but are actually biologically insignificant. Thus, we use the DUST algorithm to filter out these low complexity sequences; therefore, the algorithm displays sequences that are high-scoring and biologically significant.

Traditionally, DUST is a heuristic algorithm that takes advantage of a scoring function based on counting nucleotide triplet frequencies in 64-base windows [2]. We can use the equation given in [2] to calculate the DUST score for each 64-bit window:

$$S(a) = \frac{\sum_{t \in R} c_t(a)(c_t(a) - 1)/2}{(\ell - 1)}, \qquad (1)$$

where S(a) is the score of any sequence '*a'* with length n that is greater than 2. It has to be greater than 2 because we are using a triplet scoring function. Additionally, a sequence of length n will contain exactly n-2 triplets. For this equation, let R be the set of all 64 possible triplets using the four DNA bases A, C, T, and G. Let $c_t(a)$ be the number of times a specific triplet with value t, where t is within R, appears in *'a'*. Lastly, the variable ℓ stands for the number of triplets in '*a'*, so $\ell = n - 2$ [2].

For our implementation of the DUST algorithm, instead of using a window size of 64, we just implemented the filter with our query words (obtained by breaking up the entire query into subsequences of length 11) as any sequence where n > 2 satisfies the scoring function. Notice that the greater the frequency of recurring triplets, the higher the output score for the DUST scoring function. We filtered out all the query words that scored above a certain threshold T, which is taken in as an input argument by the BLASTn algorithm.

C. Hardware Design

For the hardware design, two main components are required: a computer with the BLASTn algorithm, and an FPGA (or array of FPGAs) for the acceleration of the Smith-Waterman portion of Blastn. For our test implementation, we are currently using the Nexys-4 Artix 7 board that we commonly used in our Verilog and VHDL classes. The advantage of this board is that it has a dedicated UART chip with a USB-UART bridge necessary for serial communication between our PC and FPGA. Our hardware implementation is designed to have the FPGA calculate the Smith-Waterman portion of BLASTn to its speed limitations. These speed limitations are bottlenecked by either the speed of the FPGA, or the speed of the serial communication between the FPGA and PC. Therefore, if given a stronger FPGA or array of FPGAs, the Smith-Waterman algorithm would be accelerated even further. In the future, other portions of Blastn can also be implemented for hardware acceleration. The Smith-Waterman algorithm acceleration was our focus because it's responsible for a minimum of 70% of our runtime (depending on query length / subject size).

1) Score Matrix

Our score matrix went through several design iterations as we had a few innovations that could result in a speedup. Ultimately, we decided to implement the design inspired by [32]. The design was put together in parts made of combinational circuits. This ensured that the speed of the Smith-Waterman algorithm was based on the propagation delay of the circuit elements within the score matrix. In order to find a score using this method, we implemented the score counter which is enabled while the score matrix is being filled, and stops after the matrix has been filled. This score, although not the same as the original Smith-Waterman algorithm, gives a score which is relative to the length and similarity of the inputs. The paper [32] goes into further detail in regard to how this scoring process works.

We began by breaking apart the score matrix into smaller components. Individual cells combine together to make up the score matrix. Each cell returns the highest score after comparing the up, left, and diagonal scores. The up and left scores were handled by the gap adder submodule as their implementations are similar, and the diagonal score was handled by the align score submodule. Gap adder simply works by using a truth table to calculate the score as we only had a few possible inputs and outputs. For align score, we had to implement three more submodules: the comparator, selector, and selector adder. As its name suggests, the comparator compares the query and the subject to find out if the corresponding letters are a match, mismatch or gap. The selector follows the comparator and assigns the match, mismatch, or gap value depending on the result of the comparator. The selector adder adds the score from the selector with the diagonal, up, and left values to calculate the diagonal score for the cell. To test our score matrix, we designed unit tests for each individual submodule and then also performed

integration testing to ensure all components work together properly.

2) UART Interface

To interconnect the software version of the BLASTn algorithm with an FPGA, a UART communication circuitry was implemented. The UART protocol was chosen as it can send data fast enough for our purposes and it is a simple enough system so that it can easily be designed to interop with our C++ system. The UART circuits, along with the Mealy finite state machines (FSMs) controlling the individual UART receiver and transmitter (separately), were designed and implemented using VHDL within the Vivado IDE. Mealy FSMs were required in order for the FPGA to be capable of transmitting/receiving multiple bytes, one at a time (since it is a form of serial communication); the stand-alone circuitries of the UART receiver and transmitter are only capable of receiving/transmitting a single byte. Therefore, the FSMs are used to control and coordinate the FPGA in a manner that not only allows it to receive multiple bytes from a computer's serial port, but process the data it receives (through the Smith-Waterman score matrix in this case) and then trigger the FPGA to send a 4 byte representation of the resulting Smith-Waterman score back to the computer. This is necessary since we are transmitting at least 14 bytes for each iteration of the algorithm. The first set of data which is transmitted from the computer (i.e. the software version of BLASTn algorithm) to FPGA is the size of the query and subject, in regards to the amount of characters within the sequences, as 4 bytes. After the FPGA successfully receives the first 4 bytes, which is tracked by a variable within the FSM, the FPGA moves into the next state and collects another 4 bytes, which are individually received by the respective UART module. The second set of 4 bytes received by the FPGA are used to represent the starting index of gaps within the query sequence(s). The FSM then moves into the next state which performs a similar function; the FPGA waits in this state until another 4 bytes are received by the UART circuit and uses these 4 bytes as the amount of gaps within the respective query sequence (beginning from the start index, which was collected in the previous state). These 12 bytes are always first since this data is used by the FPGA to recognize when the computer is finished transmitting all the data for the current inputs; subject and query sequences can have a variety of lengths, which means that the FPGA needs to know the size of each input sequence so that it recognizes when all the data for the current input has been completely received. The amount of bytes received by the FPGA in the final two states is dependent on the size, which is defined in the first four bytes received in the first state; based on the size of the query and subject, which represents the amount of characters within the sequences, the FPGA remains in these states until it collects the appropriate amount of bytes needed for that specific sequence size. The amount of bytes required for the sequence size is equivalent to the size multiplied by two, since each character is bit packed as two bits; please refer to Fig. 3 on the following page to see how each nucleotide character was bit packed in the software version of the BLASTn algorithm.

Once the FPGA has finished collecting all the data for the respective query and subject sequences, the data is then passed into and processed by the Smith-Waterman matrix module, discussed in the previous section; the score from the matrix is then transmitted back to the computer by utilizing the UART transmitter circuitry and a different Mealy FSM, in order for the FPGA to be capable of transmitting more than a single byte. A signal, which is raised for a single FPGA clock cycle when the UART is finished transmitting a single byte, is utilized to move between each state of the FSM (i.e. when the signal is high, move to the next state). Through this signal, and another which indicates when the transmitter is ready to start transmitting a new byte, the FPGA is able to send multiple bytes (one at a time) and return the output score of the Smith-Waterman matrix, within a 4 byte representation.

IV. EVALUATION

First we will evaluate our Python implementation of the BLASTn algorithm. Our Python implementation was more concerned with prototyping and the interaction between the different BLASTn modules. As a result, we emphasized implementation accuracy over speed. Thus, with all of our test data, our Python code took around 35 minutes to run; however, it produced results that validated our prototype. This allowed us to use our Python implementation as the shell for our C++ implementation.

The first version of our C++ implementation took approximately 31 minutes to fully execute. In order to speed up the program, we looked into optimizing our code base. Using the open source application Valgrind, we were able to find, test, and

resolve issues with performance and memory usage throughout our program. In order to facilitate our testing process, we used a slightly smaller dataset from here on out until we reached the final version of our C++ implementation. To improve performance and efficiency of the program, we employed smarter use of heap memory allocation and the use of parallel hashmap rather than the C++ Standard Template Library (STL) unordered_map. After resolving the inefficiencies in our code, running the program with our smaller dataset took approximately 1 minute and 5 seconds (total duration 1:05). We realized we could further speedup the program if we took advantage of compiler optimization techniques. Using C++ compiler optimization settings, our overall program runtime was reduced to 8.6 seconds with the small dataset and 8 minutes and 30 seconds (total duration 8:30) with the large dataset. At this point, we decided to move on to our hardware implementation of the Smith-Waterman algorithm as it was our bottleneck.

After overcoming a myriad of difficulties, our hardware implementation successfully ran the BLASTn algorithm with the small data set in 6041 seconds, or 1 hour and 41 minutes. We set the baud rate of our program to 1 million which means the bit rate was one million bits per second since we are dealing with binary data. We used 2 bits to represent each of the DNA nucleotide characters in an attempt to save memory, since this would mean each character is represented by 2 bits as opposed to the 8 bits, or a byte, if the characters remained in their natural type. See Fig. 3.

Fig. 3. DNA Character Representation

$A := 00$	$C := 01$	$G := 10$	$T := 11$

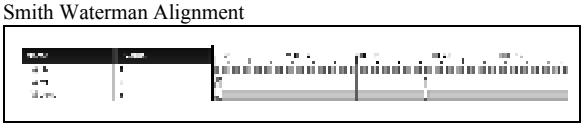
Theoretically, with a baud rate of 1 million bits per second, the FPGA should be able to transfer up to 500,000 characters per second $(1,000,000\ bps / 2\ bpchar)$. We determined our bottleneck to be the byte packing process, as we have to bit-pack each letter every time our hit is extended. While our bit packing saved memory, it cost us speed as our hardware implementation was remarkably slower than its software counterpart.

V. FUTURE WORK

Future work would entail implementing the entire BLASTn algorithm on the FPGA. Although the majority of performance gain can be accomplished by hardware accelerating the Smith-Waterman portion of the algorithm, there is value in having the entire algorithm running end to end on the board. The bottleneck in the process is the transmission rate of data between the computer and the FPGA, something that could be solved in two different ways. The first is to have the entire algorithm run on the FPGA. The second would be to use a faster communication method, such as PCIE. Additionally, there is still some benchmarking that could be done to show the true speed of our implementation. More runtimes could be tested by testing our algorithm on multiple FPGAs, and even creating arrays of FPGAs to try to see how fast our version of BLASTn could run. Another improvement to this project could be on the implementation of the Smith-Waterman algorithm itself. During our hardware implementation of the Smith-Waterman, we came up with a few different promising ideas that could result in a speed-up. Unfortunately, we had to abandon these ideas in the interest of time and choose our stable, but slower, implementation of Smith-Waterman (that was successfully implemented on the FPGA).

Additionally, during our implementation of the Smith Waterman Algorithm in VHDL, we were unable to properly simulate propagation delay. Unfortunately, the digital logic does not capture the delays that would occur in our real-world implementation. This could be addressed by an FPGA implementation that accounts for analog properties such as propagation delay between gates. As a result, our simulation displays an immediate reaction and does not properly illustrate the implementation we developed. As seen in Fig. 3, the signals do not capture any delays between pulses.

Fig. 4. Vivado Testbench Simulation for the scoring matrix in our Smith Waterman Alignment



Our hardware implementation could be greatly sped up if we implement alternatives to bit packing. We could remove it entirely, which would cost us some memory, but provide us with much greater speed, or explore alternate solutions to save memory without sacrificing speed.

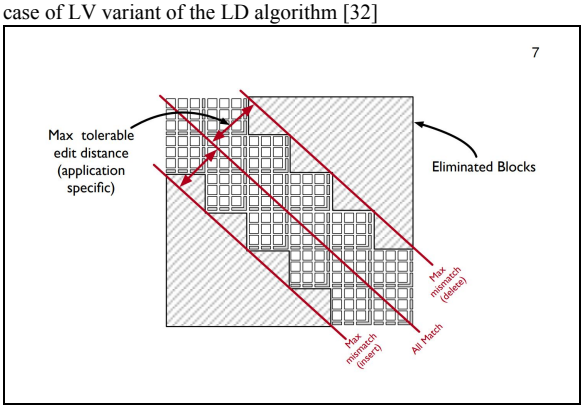
VI. RELATED WORK

Our DUST implementation was facilitated by reference [2]. Not only did it prove to be a great source of information on the original DUST

algorithm, but it also proposes SDUST which is a fast and symmetric DUST algorithm. It is an improvement on the old algorithm in that it is symmetric and will produce the same results even if the sequence is reversed, and is also not context sensitive; it is not affected by the sequences that flank it [2]. It is a faster, albeit more complicated algorithm than the regular DUST.

For our Smith-Waterman implementation, we referenced some research done by our advisor and professor, Dr. Mohamed El-Hadedy. His research, entitled "ASAP: Accelerated Short-Read Alignment on Programmable Hardware", attempts to accelerate the Smith-Waterman algorithm by reducing the amount of computation required compared to the normal Smith-Waterman calculation [32]. It does so by only calculating the matrix along the main diagonal of the matrix. Below is a picture from Dr. Aly's research in [32] illustrates the calculation along the main diagonal:

Fig. 5. Elimination of unused tiles from the ASAP lattice in the case of LV variant of the LD algorithm [32]



We referenced this idea and began to implement it, but due to time constraints, the final implementation was done with calculating the full matrix (that we already had done) for the Smith-Waterman calculation.

VII. CONCLUSION

When starting our project, we planned on developing a solution that would revolutionize the bioinformatics industry. We wanted to develop a solution to a problem that was new and currently impacting big data. Through working on this project, we learned more about the bioinformatics industry and were given the opportunity to hear the pain points experienced by researchers. As our project came to a conclusion, we learned about BLASTn and its impact in searching for similarities between a query and subject. Additionally, we learned where bottlenecks were occurring within the BLASTn algorithm and what area(s) should be targeted for optimization. The final implementation for this project was the hardware acceleration, on a Nexys 4 DDR FPGA, of the Smith-Waterman algorithm, in addition to the completed software version of the BLASTn algorithm.

VIII. REFERENCES

[1] Y. Turakhia, J. Zheng, G. Bejerano and W. Dally, "Darwin: A Hardware-acceleration Framework for Genomic Sequence Alignment", 2017. Available: https://www.biorxiv.org/content/biorxiv/early/2017/01/24/092171.full.pdf.

[2] A. Morgulis, M. Gertz, A. Schaffer and R. Agarwala, "A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences", *Journal of Computational Biology*, vol. 13, no. 5, pp. 1028-1040, 2006. Available: http://kodomo.fbb.msu.ru/FBB/year_10/ppt/DUST.pdf.

[3] J. McEntyre and J. Ostell, *The NCBI Handbook*, 1st ed. Bethesda: U.S. National Library of Medicine, 2002.

[4] R. Edwards, *BLAST*. 2012. [Accessed 20 March, 2019].

[5] Ying Chen, Weicai Ye, Yongdong Zhang, Yuesheng Xu, High speed BLASTN: an accelerated MegaBLAST search tool, *Nucleic Acids Research*, Volume 43, Issue 16, 18 September 2015, Pages 7762–7768, https://doi.org/10.1093/nar/gkv784

[6] Castells-Rufas, D. (2018, June). *Programming FPGAs with OpenCL*.

[7] Demirsoy, S. (2013). *How OpenCL enables easy access to FPGA performance?*.

[8] Desh Sing, "Higher Level Programming Abstractions for FPGAs using OpenCL," Toronto Technology Center, Altera Corporation, 2011.

[9] W. Pearson, "Selecting the Right Similarity-Scoring Matrix", *Current Protocols in Bioinformatics*, pp. 3.5.1-3.5.9, 2013. Available: 10.1002/0471250953.bi0305s43 [Accessed 2 April 2019].

[10] B. Solomon and C. Kingsford, "Fast search of thousands of short-read sequencing experiments", *Nature Biotechnology*, vol. 34, no. 3, pp. 300-302, 2016. Available: 10.1038/nbt.3442 [Accessed 2 April 2019].

[11] M. Herbordt, J. Model, B. Sukhwani, Y. Gu and T. VanCourt, "Single pass streaming BLAST on FPGAs", *Parallel Computing*, vol. 33, no. 10-11, pp. 741-756, 2007. Available: 10.1016/j.parco.2007.09.003 [Accessed: 1- April- 2019].

[12] Enormandeau. "Enormandeau/ncbi_blast_tutorial." *GitHub*, 21 April, 2016, github.com/enormandeau/ncbi_blast_tutorial.

[13] "Introduction to High Level Synthesis with Vivado HLS." *Xilinx*, 2013, users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf.

[14] "Basic HLS Tutorial using C++ language and Vivado Design Suite to design two frequencies PWM modulator system" *So-Logic*, 19 Dec. 2018,

https://www.so-logic.net/documents/upload/Basic_HLS_Tutorial.pdf
[15] “Basic HLS Tutorial using C++ language and Vivado Design Suite to design two frequencies PWM modulator system” *So-Logic*, 19 Dec. 2018, https://www.so-logic.net/documents/upload/Basic_HLS_Tutorial.pdf
[16] “Smith-Waterman Algorithm - Local Alignment of Sequences,” Amrita Vishwa Vidyapeetham. [Online]. Available: http://vlab.amrita.edu/?sub=3&brch=274&sim=1433&cnt=1. [Accessed: 26- March- 2019].
[17] B. John and T. Benos, "Scoring Matrices for Sequence Comparisons", Carnegie Mellon University, 2015. Available: https://www.cs.cmu.edu/~02710/Lectures/ScoringMatrices2015.pdf
[18] Wang, X. Xie and J. Cong, "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms." Available: https://vast.cs.ucla.edu/sites/default/files/publications/ipdps-submission.pdf. [Accessed 19 March 2019].
[19] C. Rauer and N. Finamore, "Accelerating Genomics Research with OpenCL and FPGAs", 2016. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01262-accelerating-genomics-research-with-opencl-and-fpgas.pdf. [Accessed 22 March 2019].
[20] Dai, Steve, et al. “Vivado HLS Tutorial .” Cornell.edu, www.csl.cornell.edu/. http://www.csl.cornell.edu/courses/ece5775/pdf/lecture02.pdf
[21] Xilinx, Vivado Design Suite Tutorial High-Level Synthesis, 2014.
[22] Intel, Intel High Level Synthesis Compiler, 2019.
[23] Torbjørn Rognes and Erling Seeberg, “Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors,” Institute of Medical Microbiology, University of Oslo, *The National Hospital*, Oslo, Norway. Oxford University Press, 2000.
[24] Eduard, Ayguade, Juan J. Navarro, and Dani Jimenez-Gonzalez, AMPP 0708-Q1. Class Lecture, Topic: “Smith-Waterman Algorithm.” Universitat Politècnica de Catalunya, Barcelona, Spain, October 4, 2007.
[25] Don-hyeon Park, Jonathan Beaumont, and Trevor Mudge, “Accelerating Smith-Waterman Alignment workload with Scalable Vector Computing,” University of Michigan, Ann Arbor, Michigan, September 2017.
[26] Łukasz Ligowski and Witold Rudnicki, “An Efficient Implementation of Smith Waterman Algorithm on GPU using CUDA for Massively Parallel Scanning of Sequence Databases,” University of Warsaw, Warsaw, Poland, October 2009.
[27] Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot, “Acceleration of the Smith-Waterman Algorithm using Single and Multiple Graphics Processors,” University of Massachusetts, Amherst, Massachusetts.
[28] Liang-Tsung Huang, Chao-Chin Wu, Lien-Fu Lai, and Yun-Ju Li, “Improving the Mapping of Smith-Waterman Sequence Database Searches onto CUDA-Enabled GPUs,” Department of Medical Informatics, Tzu Chi University, Hualien, Taiwan, and Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua, Taiwan, June 8, 2015.
[29] Chen B., Xu Y., Yang J., Jiang H. (2010) A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform. In: Hsu CH., Yang L.T., Park J.H., Yeo SS. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2010. Lecture Notes in Computer Science, vol 6081. Springer, Berlin, Heidelberg
[30] Lapidus, Alla. “Bioinformatics and Its Applications.”
[31] Hudson, D. “DNA Sequencing.” Bioinformatics I, pp. 145–167.
[32] Banerjee, Subho S., El-Hadedy, M., Bin Lim, J., Kalbarczyk, T. Z., Chen, D., Lumetta S. S., Iyer, R. K. (2018 May). “ASAP: Accelerated Short-Read Alignment on Programmable Hardware.”