Master in Electronic Engineering

Master's Thesis

Genomic Co-processor for Long Read Assembly

Advisor:

Prof. Marco D. Santambrogio (Politecnico di Milano)

Co-advisor:

Prof. Jordi Cosp Vilella (Universitat Politècnica de Catalunya)

Author:

Marina Martí Quesada

Barcelona, June 2021

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

telecos
BCN

Abstract

Genomics data is transforming medicine and our understanding of life in fundamental ways; however, it is far outpacing Moore's Law. Third-generation sequencing technologies produce 100X longer reads than second generation technologies and reveal a much broader mutation spectrum of disease and evolution.
However, these technologies incur prohibitively high computational costs. In order to enable the vast potential of exponentially growing genomics data, domain specific acceleration provides one of the few remaining approaches to continue to scale compute performance and efficiency, since general-purpose architectures are struggling to handle the huge amount of data needed for genome alignment.

The aim of this project is to implement a genomic-coprocessor targeting HPC FPGAs starting from the Darwin FPGA co-processor. In this scenario, the final objective is the simulation and implementation of the algorithms described by Darwin using Alveo boards, exploiting High Bandwidth Memory (HBM) to increase its performance.

Acknowledgements

I take this section to express my gratitude to several people who have helped me and given me their support during the development of this project.

First of all, I want to thank Professor Marco D. Santambrogio, advisor of this project, for giving me the opportunity to write my Master's Thesis in Politecnico di Milano.

In addition, I would like to thank Alberto Zeni, PhD Student in the NECSTLab of Politecnico di Milano, for providing me all the necessary information to start the project, mentoring me during all the semester, and for all his help and support during the realization of this project.

I express my gratitude also to Emanuele Del Sozzo and Davide Conficcioni, also PhD Students in the NECSTLab.

Also, I would like to thank Jordi Cosp for his support in this thesis and in other projects developed during the Bachelor's and Master's.

Additionally, I would like to thank Romà Macario, Asier Zubeldia, Sílvia Farràs, Jia Puig, Jenthe Goossens, Nahida Aktar, Franzi Andreas and Nesrin Kerschek for their support and their advice.

Finally, I would like to thank my parents and sister for their support since I started my university studies until now.

Thank you all very much.

Table of contents

Abstract 1
Acknowledgements 2
Table of contents 3
List of Figures 5
List of Tables 7
1. Introduction 8
1.1. Genome Assembly 8
1.2. Statement of purpose 11
2. State of the art 13
2.1. Smith-Waterman Algorithm 13
2.2. Genome alignment heuristics 16
2.3. Darwin co-processor 18
2.3.1. D-SOFT Algorithm 18
2.3.2. GACT Algorithm 19
2.3.3. Accelerator Design 20
2.4. Hardware Acceleration 23
3. Methodology 26
3.1. AXI Interface 26
3.1.1. AXI Transactions 28
3.1.2. AXI4-Lite 30
3.1.3. AXI interface implementation 30
3.2. Block Design 33
3.3. Simulations 36
3.4. High Bandwidth Memory (HBM) 38
3.5. Host Processor 42
4. Results 43
5. Budget 45
6. Conclusions and future development 48
Bibliography 49
Appendices 51

List of Figures

Figure 1. Genomics Data growth comparison with Moore's Law and CPU performance from 2000 to 2018. 8

Figure 2. Genome assembly example....9

Figure 3. Example of input genome sequence (orange), its copies (blue), and its set of reads (green)....9

Figure 4. Example of a set of ordered reads from an input sequence (left), and the actual reads obtained from the input, with unknown order (right)....10

Figure 5. Sequence alignment scoring scheme example. 13

Figure 6. Four macro phases of the Smith-Waterman algorithm. 14

Figure 7. Smith-Waterman matrix calculation example. 15

Figure 8. Smith-Waterman traceback matrix example....16

Figure 9. Illustration of D-SOFT algorithm for k=4, N=10, h=8, NB=6....19

Figure 10. Illustration of extension stage in GACT algorithm using an example dynamic programming (DP) matrix for parameters (T=4, O=1). 20

Figure 11. Reference-guided and de novo assembly (overlap step) using D-SOFT and GACT. 20

Figure 12. Darwin's systolic array architecture....21

Figure 13. Darwin's systolic array architecture with traceback storage....22

Figure 14. Comparison between the X-drop algorithm and the GACT algorithm. 22

Figure 15. Darwin's outer-loop parallelism. 23

Figure 16. Host CPU augmented with Xilinx FPGAs in Alveo Cards. 24

Figure 17. Software stack (left) and hardware components (right) communication through PCIe. 24

Figure 18. Alveo card shell and role topology. 25

Figure 19. High-Level Diagram of Two HBM Stacks. [2] 26

Figure 20. Basic handshake mechanism of the AXI protocol. 27

Figure 21. AXI Read Address and Read Data channels. 28

Figure 22. AXI Write Address, Write Data and Write Response channels....28

Figure 23. AXI Read transaction example....29

Figure 24. AXI Write transaction example....30

Figure 25. Top-level Darwin kernel structure....31

Figure 26. Darwin kernel RTL block structure. 32

Figure 27. Function prototype and register map for the Darwin kernel. 34

Figure 28. Top-level blocks of the design (control register and Darwin). 35
Figure 29. Darwin Block design. 36
Figure 30. Query and reference sequences used for simulation. 36
Figure 31. Simulation results for the query “ATGCT” and reference sequence “AGCT”. .37
Figure 32. AXI Master 0 interface signals in Vivado Simulation. 37
Figure 33. Darwin Logic simulation results for Q = “ATGCT” and R = “AGCT”. 38
Figure 34. Alveo U280 board with the two HBM stacks...39
Figure 35. Architecture of Alveo HBM subsystem. 40
Figure 36. Platform diagram in Vitis Analyzer tool..40
Figure 37. Information about the kernel bitstream generation, provided by Vitis. 41
Figure 38. Darwin Kernel system diagram obtained through Vitis HLS. 41
Figure 39. GACT array for tile size T = 320...43

List of Tables

Table 1. Short and long read assembly required hours. 10
Table 2. AXI configurations used for each GACT input in the RTL kernel. 33
Table 3. Resource utilization of the final design implemented in the Alveo U280. 43
Table 4. Design timing summary...43
Table 5. Temporal analysis of the project. 45
Table 6. Breakdown of the project budget...46

1. Introduction

Genomics data is transforming medicine and our understanding of life in fundamental ways; helping the diagnosing and decoding of diseases, the understanding of ancestry through genotype-phenotype matching, understanding molecular bases of evolution, and much more. However, from 2001 to 2018, the growth of genomics data has been of 125% per year, and it is far outpacing the Moore's Law. In contrast, CPU performance has slowed down significantly; from 20% per year in early 2000's to only 3% per year today, which results as a considerable gap between CPU performance and genomics data. This gap is expected to keep increasing in the future, as genomics data continues to double every year, as shown in figure 1.

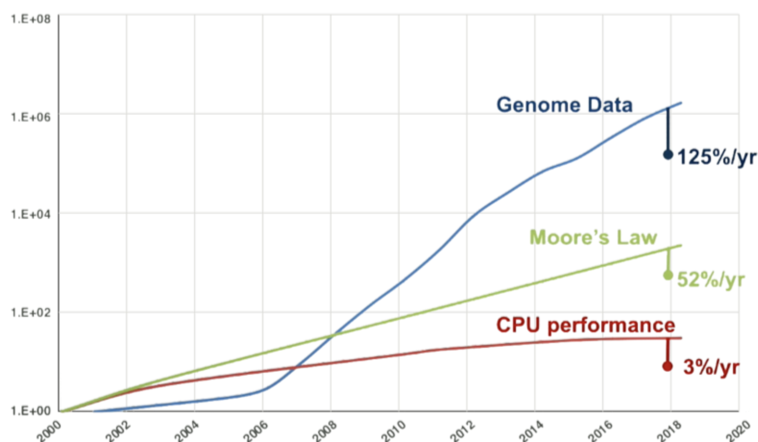

Figure 1. Genomics Data growth comparison with Moore's Law and CPU performance from 2000 to 2018.

1.1. Genome Assembly

Genome assembly is the process where many copies are chopped up into fragments from a genome, then these fragments are sequenced, and finally they are put together to obtain the original genome from those reads previously sampled.

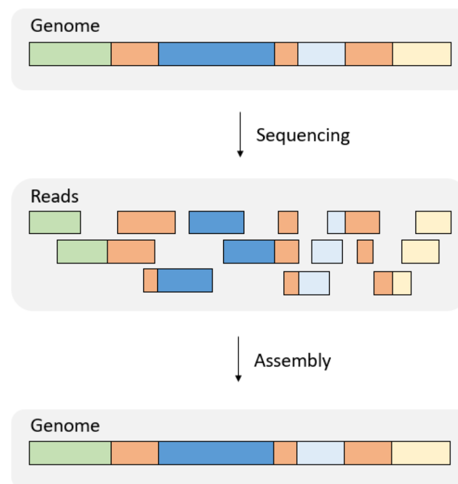

Figure 2. Genome assembly example.

As figure 2 shows, genome assembly can be understood like putting together a puzzle without being able to know what it looks like. The input of the assembly process is a genome sequence, for example, the sequence of the figure 3 (in orange). As shown, a genome sequence consists of a list of nucleotides (A, C, G and T for DNA genomes) that make up all the chromosomes of an individual or a species. From that input sequence, many copies are taken and fragmented into pieces, which are then read using DNA sequencers. This process is called whole-genome “shotgun” sequencing, where “shotgun” refers to the random fragmentation of the whole genome, like it was fired from a shotgun.

Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Fragment: GGCGTCTA TATCTCGG CTCTAGGCCCTC ATTTTTT
GGC GTCTATAT CTCGGCTCTAGGCCCTCA TTTTTT
GGCGTC TATATCT CGGCTCTAGGCCCT CATTTTTT
GGCGTCTAT ATCTCGGCTCTAG GCCCTCA TTTTTT

Figure 3. Example of input genome sequence (orange), its copies (blue), and its set of reads (green).

The problem is that the reads are not known where they come from, we do not have the total ordering of the reads, only having the read sequences, the assembly process has to figure out what the order of those reads were on the original genome, like shown in figure 4.

Figure 4. Example of a set of ordered reads from an input sequence (left), and the actual reads obtained from the input, with unknown order (right).

There are two types of genome assembly; referenced assembly and *de novo* assembly. In the referenced assembly, two input sequences are used, the reference and the query, where the reference sequene give an approximation of the ordering of the original genome. On the other hand, in *de novo* assembly nothing is known about the original genome, only having the set of reads, and the original sequence has to be reconstructed from the query sequence.

The target application of this work is genomics assembly. The reason why we have such a massive growth in genomics data is due the technology of DNA sequencing, which enables us to read raw nucleotides from a DNA sample. However, the result sequences can be different from the original sequence from which the read was taken, and in particular, the read will have a certain number of substitutions, deletions or insertions with respect to the original sequence, and that is what we call alignment. Most of these errors tend to be stochastic in nature, and so we can naturally lower the errors or fix them, using a consensus of multiple alignment reads, and this process is what we call assembly.

As sequencing technologies incur prohibitively high computational costs, in order to enable the vast potential of exponentially growing genomics data, domain specific acceleration provides one of the few remaining approaches to continue to scale compute performance and efficiency, since general-purpose architectures are struggling to handle the huge amount of data needed for genome alignment.

Long reads tend to be very noisy, with a 15-40% of error rate, compared to 0-2% error rate in short reads. As a result, the assembly process, typically when using a reference sequence to guide the assembly, takes up to 5000 CPU hours, and when we assemble the genome *de novo* without the bias or help of a reference genome it can take upt to 60000 CPU hours, which is orders of magnitude higher than in short read sequencing. This is the reason why the acceleration for long read assembly is needed.

Table 1. Short and long read assembly required hours.

	Reference-guided assembly (54x human)	**De novo assembly (54x human)**
Short reads (~100bp, 0-2% error rate)	Up to 200 CPU hours	Up to 2,000 CPU hours
Long reads (~10Kbp, 15-40% error rate)	Up to 5,000 CPU hours	Up to 60,000 CPU hours

Long reads are considered to be the holy grail of sequencing for many reasons. It is the only technology which enables us to read a wide spectrum of mutations, in particular structure variations, which is tandem duplications, inversions or pseudo gene insertions in haplotype phasing, which is to distinguish maternal a paternal mutations, and in resolving the repeats, since around 50% of the human genome is repetitive.

In 2008-2009, sequencing technologies that took a different approach than second-generation platforms were dubbed "third-generation."

Pacific Biosciences and Oxford Nanopore Technology are two businesses that are at the forefront of third-generation sequencing technology development. When it comes to sequencing single DNA molecules, these businesses take very different approaches.

PacBio invented the real-time sequencing platform for single molecules (SMRT).

Passing a DNA molecule through a nanoscale pore structure and then monitoring changes in the electrical field surrounding the pore is how Oxford Nanopore's technology works.

1.2. Statement of purpose

The aim of this project is to implement a genomic co-processor targeting high performance computing (HPC) FPGAs, starting from the Darwin co-processor, previously developed by Yatish Turakhia, Gill Bejerano and William J. Dally, from Standford University in 2018 [1]. Darwin is a co-processor for genomic sequence alignment that provides up to 15,000x speedup over the state-of-the-art software for reference-guided assembly of third generation reads. Darwin achieves this speedup through hardware-software co-design, by combining a novel filtering algorithm, D-SOFT, and a hardware-accelerated version of GACT, a novel alignment algorithm. Darwin has been implemented for their developers in an Intel Arria 10 FPGA. In this project the target hardware is HPC FPGAs, specifically the Xilinx Alveo boards [2].

Field Programming Gate Arrays (FPGAs) are a particular family of integrated circuits intended for custom hardware implementation, with the key property of being capable of reconfiguration for an infinite number of times. Currently FPGAs are the state of the art of Programmable Logic Devices. Reconfiguring an FPGA means changing its functionality to support a new application, and it is equal to have some new pieces of hardware mapped on the FPGA chip, having to implement a new functionality. In other words, FPGA make it possible to have custom-designed high-density hardware in an electronic circuit, with the added bonus of having the possibility of changing it whenever there is the need, even while the whole application is still running. Nowadays we can find FPGAs in the cloud, like in the Amazon F1 instances, moreover, the “incorporation” of reconfigurable array logic into a microprocessor provides an alternative growth path that allows application specialisation while benefiting from the full effects of commoditisation. Like modern reconfigurable logic arrays, a single microprocessor design can be employed in a wide variety of applications. Application acceleration and system adaptation can be achieved by specialising the reconfigurable logic in the target system or application. This has led to a new concept for computing: if a processor can be coupled with one or more FPGA-like devices, it could in theory support a specialised application specific circuit for each program, or even for each stage of a program’s

execution. The unlimited reconfigurability of an FPGA permits a continuous sequence of custom circuits to be employed, each optimised for the task of the moment.

There are several accelerators that can be used: examples are GPUs and FPGAs. On one hand, GPUs offer better parallel performance, more efficient computing and an easy to use programming model. On the other hand, FPGAs bring higher performance-per-watt, improved hardware acceleration performance, and lower inter-device latency. Moreover, when used in cloud infrastructures, the advantages introduced by extending these infrastructures with Xilinx FPGAs technologies are even more clear. In this context FPGAs are essentially a systems-on-a-chip coupled with one or more host CPUs via a PCI Express connection. This scenario is basically turning the FPGA into a custom accelerator for the code running on the CPU. This is extremely interesting and convenient because FPGA resources are ready to be used upon purchase and can be elastically scaled.

The huge amount of data they need to process and the complexity of the genomic algorithms, have raised the problem of increasing the amount of computational power needed to perform the computation. In this scenario, hardware accelerators reveal to be efficient in achieving a speed-up in the computation of this algorithms, while at the same time, saving power consumption. Among the algorithms used in genomics, and in the Darwin itself, the Smith-Waterman (SW) algorithm is a dynamic programming algorithm, guaranteed to find the optimal local alignment between two nucleotide strings. Successive FPGA-based hardware acceleration in this algorithm is used to perform pairwise alignment of DNA sequences.

Comparing Intel FPGAs, where the original Darwin is implemented, to Xilinx HPC FPGAs, where the co-processor is implemented in the present work, these last ones are mode widely available since, for instance, they can be used online, available in the cloud, as mentioned before. Some examples of Xilinx FPGA-based cloud infrastructures are Baidu FPGA Cloud Server, the Huawei FPGA Accelerated Cloud Server (FACS), or the Amazon Elastic Compute Cloud (EC2 F1). Implementing Darwin in Xilinx FPGAs instead of Intel FPGAs enables the fact that the co-processor algorithm can be used on multiple architectures, and not only on Intel specific ones. Xilinx FPGAs can deliver the flexibility, application breadth, and feature velocity required by complex and constantly changing application workloads.

2. State of the art

2.1. Smith-Waterman Algorithm

Among multiple tasks performed in bioinformatics to analyse DNAs, one of the most compute intensive one is the task of identifying regions of similarity between sequences of DNA, RNA or protein, called sequence alignment. The majority of hardware acceleration efforts for genomic alignment have focused on the Smith-Waterman (SW) and Neeldman-Wunsch (NW) algorithms. These algorithms find exact pair wise alignments and have quadratic complexity in the length of the reads.

The sequence alignment problem takes a query sequence Q of letters in {A, C, G, T}, corresponding to the four nucleotide bases, and tries to find the best match (alignment) with a reference sequence R, assigning each letter in R and Q to either a single letter in the opposite sequence, or to a gap. The scoring scheme rewards matching bases, and penalises mismatches and gaps. For example:

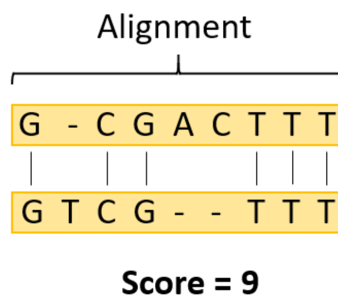

Figure 5. Sequence alignment scoring scheme example.

The SW algorithm is a dynamic programming (DP) algorithm, guaranteed to find the optimal local alignment between two genome strings or sequences. This algorithm has been firstly introduced in 1981 by Mr Smith and Mr Waterman and it is based on the previously published algorithm called Needleman-Wunsch. The SW algorithm performs local sequence alignment between two strings, usually referred as query, or read, and the reference sequences.

The SW is a DP algorithm as it decomposes a big problem into smaller ones, storing the intermediate results and using them again when solving the next sub-problem.

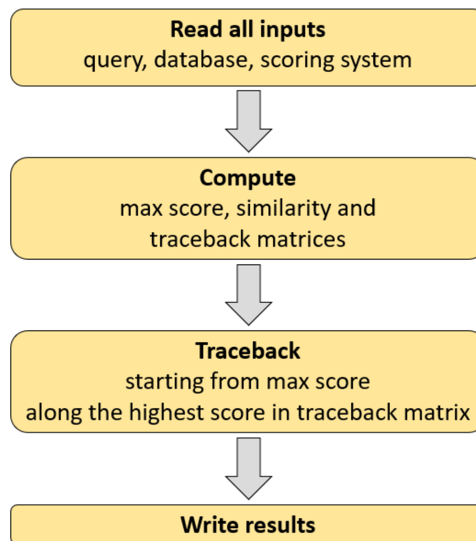

Figure 6. Four macro phases of the Smith-Waterman algorithm.

As figure 6 shows, the SW algorithm can be organised into four macro phases: Read, Compute, Traceback, and Store or Write.

Given a query Q, a reference M, a similarity system and an affine gap function, the algorithm calculates the scoring matrix S and the Traceback Matrix T, whose dimension is QxM each. Finally it traces back the path of elements starting from the index identifying the maximum element in the similarity matrix, and terminating whenever a value with value zero is found. One of the most important features of these algorithms is that it is guaranteed to find the optimal local alignment between the two strings, with respect to the scoring system that is provided as an input.

Because of the high computational needs required by the algorithm, it has been modified during the years and multiple heuristic methodologies have been introduced. These heuristics, on one hand allow to increase the overall performance of the system, but on the other hand decrease the precision of the algorithm, that does not guarantee anymore the optimality of the alignment found.
Furthermore, they may not be able to find similarities for sequences that are very distant. The right trade off in between algorithm precision and system performance has to be found. To avoid the lost in precision, several solutions have been tried over the years to achieve better performance but without loosing in precision. Because of this, in the state of the art it is possible to find multiple implementations of this algorithm using general purpose CPU, and different hardware accelerators such as Graphical Processing Unit and FPGAs.

As seen in figure 6, it is true that the SW algorithm is composed by four phases but, considering that the first one is used to read all the necessary inputs and the last one has the objective of giving back the computed results, it can be basically considered as a 2 phase, or steps, algorithm. The first step can be referred as matrices calculation phase, while the second one as traceback step.

Given a query N, a reference M, a similarity system and an affine gap function, the algorithm calculates the scoring matrix and the traceback matrix, whose dimension is NxM each. Finally, it traces back the path of elements starting from the index identifying the maximum element in the similarity matrix, and terminating whenever a value with

value zero is found. During the calculation of the similarity matrix, it is as well needed to keep track of the index of the maximum element found, as these values are going to be used for the second step of the algorithm. Each element of the similarity matrix is populated by using the relation presented in the following formula.

$$S(i,j) = \begin{cases} S(i-1,j-1) + s(N_i, M_j) & Match/Mismatch \\ \max_{k\leq 1} S(i-k,j) + gap_{del} & Deletion \\ \max_{k\leq 1} S(i,j-l) + gap_{ins} & Insertion \\ 0 & \end{cases} ,1 \leq i \leq m, 1 \leq j \leq n \quad (1)$$

Where S is used to identify the Similarity Matrix, $S(i,j)$ is the i, j element in the Similarity Matrix, $S(N_i, M_j)$ represents the similarity function over the two strings N_i and M_j, gap_{del} is the gap scoring value in case of deletion, and finally gap_{ins} is the gap scoring value in case of an insertion.

	C	G	T	G	G
G	0	2	0	2	2
C	2	2	2	2	2
G	0	3	4		
T	0				
C	2				

Figure 7. Smith-Waterman matrix calculation example.

As an example of matrix calculation, in figure 7, for each element of the matrix it is necessary to compute its three dependency values first. These dependency values are the ones highlighted in green in the figure, and are; the value above the current one, or north value, the value on the left of the one considering, or west value, and the value that is positioned in the upper left position, or north-west value.
Once these values are computed, then is possible to compute the current one by performing some basic operations. For all the cell of the matrix where there is not one of the dependency value, as for the first row and first column, the dependency values are supposed to be equal to zero. While computing the similarity matrix as described so far, it is necessary to keep a track of what is the maximum value inside the matrix, and in particular, in the index of this value, so that it can be used in the traceback.
The similarity matrix in fact, is not the only matrix that needs to be computed at this stage. It is also necessary to compute the traceback matrix in parallel to the first one. This second matrix is used to store all the directions that the second step will have to follow so that the final string can be obtained. Figure 8 shows an example of a traceback matrix.

	C	G	T	G	G
G	C	C			
C	N_W	N			
G	C	N_W			
T	N_W	N	W	N	W
C	C	N	N_W	W	W
C	C	W	N	N_W	N_W

Figure 8. Smith-Waterman traceback matrix example.

As it can be understood from the formula (1), the similarity store will hold the maximum value between zero and the three dependency values. The first one is a value composed of the sum of the element that lies on the anti-diagonal and another value that depends on the similarity function applied. The second element is the sum of the value on the left of the one considered and the gap value in case of deletion. The third value is composed of the gap value in case of insertion and the element over the one considered.
Once identified the maximum between these three values, the algorithm stores in the traceback matrix the position of the value that originated the identified maximum. In the example of the figure 8, in case of the maximum value is the first one, the program will store in the traceback matrix the value north-west (N_W), in the second case it will store west (W) and in the third one north (N). In case none of the previous is the maximum, it means that the maximum value is a zero, then the program will store centre (C).
At the end of the first step of the algorithm, two matrices are produced; the similarity matrix and the traceback matrix. After this, the second step of the computation is the traceback. This is the final step and produces the real output, the optimal local alignment between the query and the reference. This step, starts from the maximum index identified before and iteratively passes through the elements in the traceback matrix by following the stored directions. The process continues until a cell where the stored value is centre is encountered. Finally, it outputs how the query sequence aligns to the reference one. The output can be generated or provided by using multiple formats. An output example can be represented by the cigar string, which specifies the length of the bases aligned as well as the associated operation that can be base alignment, both match and mismatch, deletion and insertion of bases.

2.2. Genome alignment heuristics

Other heuristic approximations to the Smith-Waterman algorithm for pairwise alignment of sequences include Banded Smith-Waterman [3], X-drop [4] and Myers bit-vector algorithm [5]. All existing heuristics complete the matrix-fill step before starting traceback, and therefore, having a memory requirement that grows at least linearly with the length of the sequences being aligned.

Banded Smith-Waterman is an algorithm for aligning two sequences within a diagonal band that requires only O(NW) computation time and O(N) space, where N is the length of the shorter of the two sequences and W is the width of the band [3].

The X-drop algorithm [4] avoids the full quadratic cost by searching only for high-quality alignments, and can be viewed as an approach to accelerate both NW and SW. Most applications of alignment will throw out low quality alignments, which arise when the two

strings are not similar. Instead of exploring the whole NxW space, the X-drop algorithm searches only for alignments that results in a limited number of edits between the two sequences. X-drop keeps a running maximum score and does not explore cell neighbourhoods whose score decreases by a user-specified parameter X. It gets its performance benefits from searching a limited space of solutions and stopping early when a good alignment is not possible.

The Myers bit-vector algorithm [5] is a design that finds all spots where a query of length M matches a fragment of a sequence of length N with k-or-fewer variations using simple and practical bit-vector algorithms. By calculating a bit model of the relocatable dynamic programming matrix for the string-matching problem, the Myers approach requires just O(NM/w) time, being *w* the word size of the machine (e.g., 32 or 64). Thus, the algorithm performance is independent of k.

The Darwin co-processor [1], which is the starting point for the current project, and is defined with detail in the following section, uses two novel algorithms, D-SOFT and GACT. The co-processor is based on the classical *seed-and-extend* paradigm. This approach starts with substrings (the *seeds*) of fixed size *k* drawn from the query sequence (*Q*), and finds their exact matches in the reference sequence *R* (called *seed hits*). Once seed hits are found, the cells surrounding the hits are explored, using a dynamic programming approach. This avoids the high cost of exploring the full space. A drawback of the *seed-and-extend* approach is that alignments with no exactly matching substrings of length greater than or equal to *k* will not be discovered, reducing the algorithm's sensitivity.

In Darwin, the *seeding* step is handled by the D-SOFT algorithm, whereas the *extension* phase is handled by GACT. D-SOFT falls under the group of *seed-and-extend* algorithms, which were popularized by BLAST [6], and subsequent filtration techniques based on measuring the number of seed hits preserved in a band of diagonals. Two-hit BLAST, GraphMap [7], and BLASR [8] are among them.

BLAST [6] is a sequence alignment method that directly approximates alignments that optimize the maximum segment pair (MSP) score, a measure of local similarity. The core approach is straightforward and reliable; it may be implemented in a variety of ways and used in a variety of situations, such as analysing several regions of similarity in long DNA sequences.

GraphMap [7] is a mapping approach for analysing Oxford Nanopore sequencing reads that gradually refines candidate alignments to reliably handle potentially high-error rates and uses a fast graph traversal to match lengthy reads with speed and precision (>95%).

DALIGNER [9] was the first way to directly count the bases in the *seed hits* of a diagonal band, eliminating the need to multiple count overlapping bases in the hits. Darwin's D-SOFT was inspired by this algorithm. DALIGNER is a threaded filter that suggests seed points between pairs of readings that are likely to pass through a significant local alignment. D-SOFT and DALIGNED differ in their implementation. DALIGNER finds overlaps by sorting and merging large pairs of read blocks, which necessitates a large number of random accesses to off-chip memory, whereas D-SOFT performs highly sequential accesses to off-chip DRAM using a seed position table, and random accesses during filtration are handled by fast on-chip memory, making the algorithm better suited to hardware acceleration.

There are filtration approaches that are not focused on counting seeds in a band of diagonals nor the number of bases, such as BWA-MEM [10]. Instead, BWA-MEM, use super maximum seeds.
Other examples include Canu [11], M-HAP [12] and LSH-ALL-PAIRS [13], which are based on probabilistic, locality-sensitive hashing of seeds.

Prior work, like Darwin's, has concentrated on developing a complete sequence alignment framework that includes filtration as well as sequence alignment with very adjustable parameters and hardware acceleration. TimeLogic [14] is one such example. TimeLogic provides FPGA-based framework for BLAST, Smith-Waterman, HMM profile search, and gene finding algorithms, running sophisticated high-throughput and time-consuming comparisons, reducing the computational time of whole human genome to under 20 minutes.

2.3. Darwin co-processor

Darwin employs an extension approach that is the first approximate Smith-Waterman technique with a constant memory demand for the compute-intensive step, allowing for hardware acceleration of arbitrarily long sequence alignments. Furthermore, its alignments are empirically optimal when there is enough overlap.

Darwin is a long read assembly coprocessor based on hardware-software co-design. It makes use of the two innovative algorithms D-SOFT and GACT, which were mentioned earlier. The coprocessor is built on the *seed-and-extend* paradigm, in which D-SOFT does the seeding and GACT does the extension. The methods are built to take advantage of hardware acceleration.

The extension step, which is employed by the GACT technique, is more compute-intensive, whereas D-SOFT's purpose is to drastically minimize the search space for it during the alignment of a reference *R* with a query *Q*.

2.3.1. D-SOFT Algorithm

D-SOFT starts with *N* *k*-length substrings (*seeds*) selected from *Q* and finds their exact matches (*seed hits*) in *R*. *R* is pre-processed to create a seed position table, which allows for quick retrieval of seed hits for a given seed. *R* is then divided into N_B unique bins, each of which is paired with a diagonal band with a slope of 1.

The number of unique bases in *Q* covered by seed hits in each diagonal band is then counted by D-SOFT, and if the count in a given band exceeds *h* bases, the band is chosen for further extension. This procedure is shown in Figure 9 for $k = 4$, $N = 10$, $h = 8$, and $N_B = 6$.

There are six bins in all, with bins 1 and 3 having three seed hits each. However, bin 1 only covers six bases in seed hits, but bin 3 covers nine bases, implying a larger likelihood of finding a high-scoring alignment. Only bin 3 is chosen as a candidate bin because the *h* threshold is set to 8.

Because overlapping bases in seed hits are not tallied multiple times, counting unique bases in a diagonal-band allows the algorithm to be more exact at high sensitivity than strategies that merely count the number of seed hits. D-SOFT's parameters can be tweaked to match the needs of various sequencing technologies.

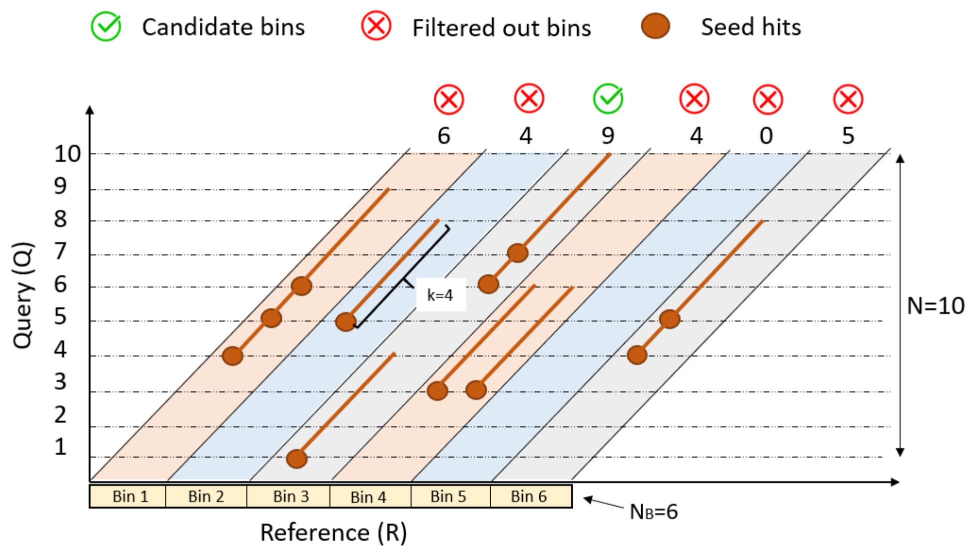

Figure 9. Illustration of D-SOFT algorithm for k=4, N=10, h=8, NB=6.

2.3.2. GACT Algorithm

In the seed-and-extend paradigm, GACT executes the extension step, in which dynamic programming (DP) is employed around the seed hit to assign gaps ("-") to *R* and *Q* in order to produce an alignment that optimizes the score. Prior heuristics lower the optimal Smith-Waterman algorithm's space and time complexity from *O(mn)* to linear *O(m+n),* where *m* and *n* are the lengths of the two sequences to be aligned.

Previous methods required prohibitive traceback pointer storage for hardware implementation, whereas long read sequencing technology requires alignment to two lengthy sequences. The GACT technique is notable for finding arbitrarily lengthy alignment extensions with a constant amount of traceback memory (as opposed to linear or quadratic in earlier algorithms), as illustrated in Figure 10, by following the optimal path among overlapping tiles of a given maximum size.

GACT traceback is a modification of the original Smith-Waterman technique that allows you to combine numerous tile alignments into a single alignment. The tile size *T* and the overlap threshold *O* are the two parameters used. With appropriate values of *T* and *O*, this algorithm produces an optimal result (similar to Smith-Waterman), requiring only T^2 pointers to be held in memory for a tile of size *T*. T_{max} = 512 base-pairs is supported by GACT hardware, which is sufficient to create empirically ideal alignments for lengthy reads. This reduces the storage required for a tile's traceback pointer to just 128 KB, allowing GACT to be massively parallelized on specialized hardware.

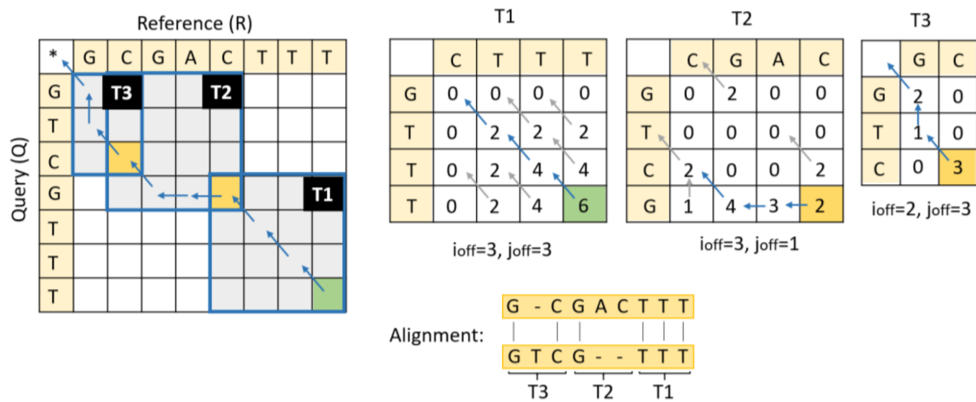

Figure 10. Illustration of extension stage in GACT algorithm using an example dynamic programming (DP) matrix for parameters (T=4, O=1).

Figure 11 depicts the use of Darwin for reference-guided and *de novo* assembly. The forward and reverse complements of *P* reads in *S = {R_1, R_2, ..., R_P}* are employed as *Q_I* queries into a reference sequence *R* in both circumstances.

In reference-guided assembly, an actual reference sequence is used, but in *de novo* assembly, Darwin speeds up the most compute-intensive overlap phase, which involves applying the $O(P^2)$ algorithm to discover pairs of overlapping reads. The last-hit position of each candidate bin is supplied to a GACT array, and *N* seeds from each query *Q_I* are fed to D-SOFT. The alignment is extended and scored using GACT.

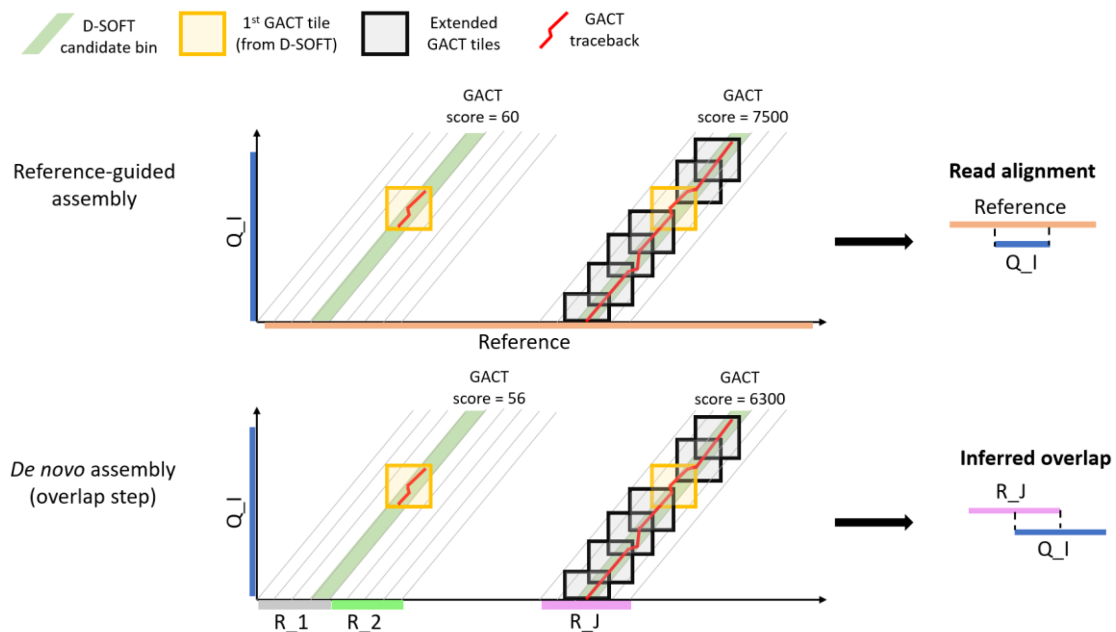

Figure 11. Reference-guided and de novo assembly (overlap step) using D-SOFT and GACT.

2.3.3. Accelerator Design

In the Darwin co-processor, D-SOFT, as explained before, implements the filtration component of the alignment, in particular is used to reduce the search space for the more

compute-intensive dynamic programming step. Whereas GACT is an algorithm which uses constant time and space *O(1)* for the most compute-intensive step that it has, in order to do arbitrary long alignments using dynamic programming, and it is heuristic to the Smith-Waterman algorithm.

Darwin's sources of acceleration can be summed up in hardware-software codesign, specialized operations, parallelism (outer and inner loop), usage of local memories, optimized memory accesses and reduced overhead.

The first step to accelerate the alignment phase is to implement specialized operations, accelerating the Smith-Waterman algorithm, computing a dynamic programming matrix between a reference and a query sequence, using the following equations.

$$I(i,j) = max\begin{cases} H(i,j-1) + gap_open \\ I(i,j-1) + gap_ext \end{cases} \quad (2)$$

$$D(i,j) = max\begin{cases} H(i-1,j) + gap_open \\ D(i-1,j) + gap_ext \end{cases} \quad (3)$$

$$H(i,j) = max\begin{cases} 0 \\ I(i,j) \\ D(i,j) \\ H(i-1,j-1) + W(r_i,q_j) \end{cases} \quad (4)$$

For insertions (*I*) and deletions (*D*), equations 2 and 3 compute the affine gap penalty. Equation 4 calculates the cell's overall score. For each cell, a four-bit traceback pointer is computed: one bit for equations 2 and 3 to indicate whether the insertion (deletion) score was obtained by opening or extending a gap, and two bits for equation 4 to indicate whether the final score was reached by null (terminating), horizontal, vertical, or diagonal cell.

Computing these equations in a CPU, they have 35 arithmetic instructions plus 15 load/store operations, and they will take 37 clock cycles because of the dependencies. However, using a specialized unit (Processing Element – PE), these operations can be done in one single cycle.

Darwin's accelerator exploits parallelism, making use of systolic array of these processing elements to exploit the wave-front parallelism that is available in the Smith-Waterman equations, having 64 PEs per array. Each PE maintains the maximum score and corresponding position for the cells it has computed. The communication is very simple, one-way nearest neighbour.

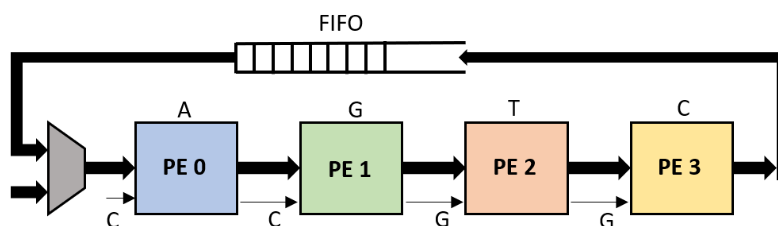

Figure 12. Darwin's systolic array architecture.

The *H* and *D* scores of equations 3 and 4 of the last PE of the array are stored in a FIFO, as shown in figure 12, and are consumed by the first PE during the computation of the next query block. The depth of this FIFO is T_{max}, which corresponds to the DP-matrix's maximum number of columns.

It is also needed to store some traceback state for each of these tiles, and so each processing element is connected to each unique bank of SRAM, which is only 2KB in size, as shown in figure 13. In this way, accessing local memories is orders of magnitude cheaper compared to accessing off-chip memory.

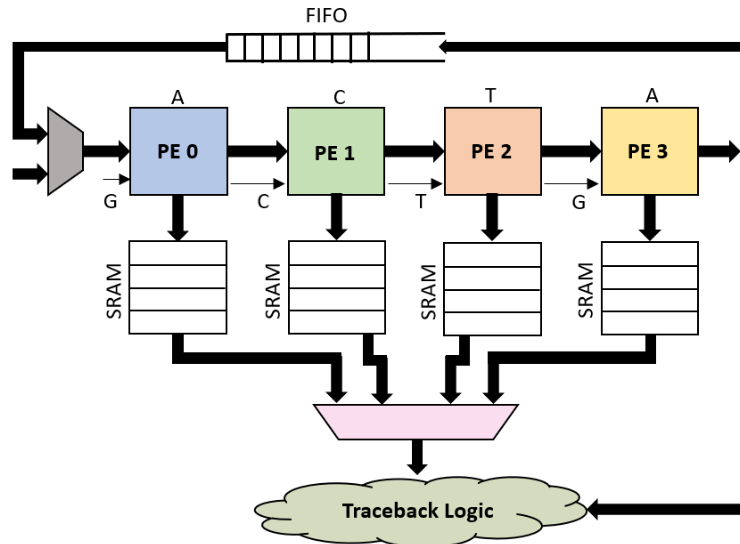

Figure 13. Darwin's systolic array architecture with traceback storage.

GACT is a classic example where co-design is used to enable local memory. Comparing the X-drop algorithm [4], which is used in BLAST [6], this would require order of length of the aligning sequences to be stored in the memory, and as a result accessing off-chip memory would be required. In GACT, by means of changing the co-design algorithm and doing the computation as a series of overlapping tiles, it is only needed to store the memory for a single tile, which is constant. Figure 14 shows the differences between the X-drop algorithm [4] and the GACT algorithm [1], the first one requiring *O(L)* space in the memory, while the second one using constant space *O(1)* to be stored.

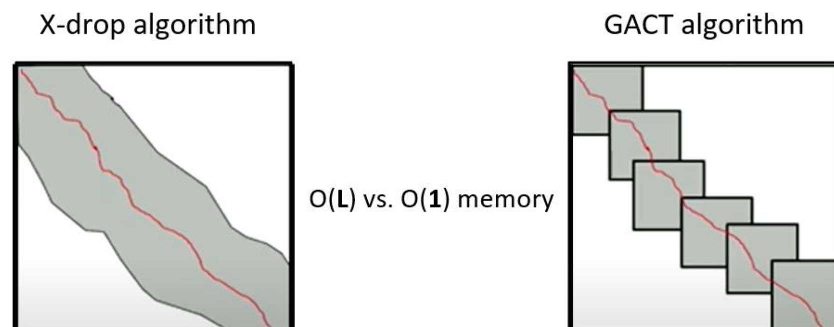

Figure 14. Comparison between the X-drop algorithm and the GACT algorithm.

Darwin also exploits outer-loop parallelism, replicating a lot of the arrays of figure 12, using 64 independent arrays, each array having 64 PEs.

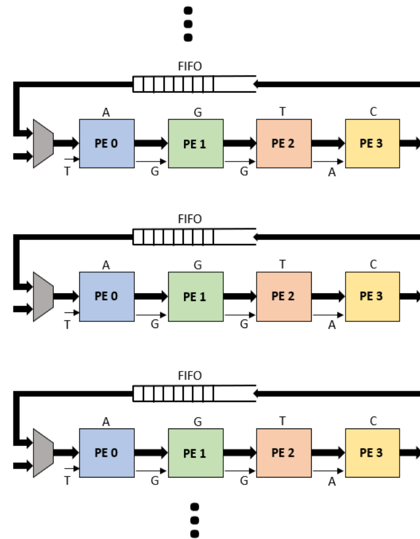

Figure 15. Darwin's outer-loop parallelism.

Increasing outer-loop parallelism results in the need of more storing states, which is what Darwin stores per tile, for that, the co-processor has a good balance between how much of outer-loop parallelism and inner-loop parallelism is exploited.

2.4. Hardware Acceleration

This project has been developed making use of the Xilinx Vitis development environment, implementing it on the Alveo u280 board. Genomic applications are ideally suited to be implemented on a system where they can gain advantages by being accelerated because of the presence of an FPGA, exploiting massive parallelism and deep pipe aligning. The Alveo cards consist of a host CPU coupled with Xilinx FPGA. In particular, the Alveo u280 instance is using Xilinx UltraScale+ XCU280-L2FSVH2892E. The Alveo u280 counts with 32-pseudo channels of High Bandwidth Memory (HBM), offering a total of 8 GB of HBM, at 460 GB/s, providing high-performance and adaptable acceleration for memory bound, compute-intensive applications, as is genomic alignment in Darwin.

With these specifications this card allows for massive IOs transfer and massive parallelism, and obviously because of the fact that it inherits all the benefits of being an FPGA device, it is ideal for accelerating compute intensive applications.

Alveo cards consist of a host CPU augmented with Xilinx FPGAs, and their acceleration works by moving compute intensive and deeply pipeline parts of application under execution to the FPGA, where customized implementation of hardware accelerator can be added to provide the needed performance boost. The Alveo instances are not providing only FPGA devices, therefore, the rest of the remaining part of the application will be executed on the x86 CPU.

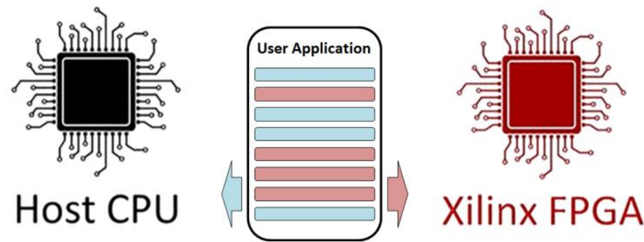

Figure 16. Host CPU augmented with Xilinx FPGAs in Alveo Cards.

As figure 17 shows, on the left side there is the software stack running on the x86 CPU (host). As seen, on the host CPU the custom application interacts with the FPGA by using the OpenCL API, and the OpenCL Runtime is responsible for managing and servicing the various acceleration requests sent to the FPGA.

The same thing can be done with the FPGA. Therefore, on the right side of figure 17, the red box represents the hardware components that are implemented on the FPGA. The communication infrastructure between the CPU and the FPGA is carried over the PCI Express interface. Within this context at the very lowest level, the drivers handle the PCI transfer between the two devices. Once the DDR or HBM memory is loaded with the necessary data, the customer isolation kernels can then read it, process the data, and write it back to the DDR or HBM using standard AXI interfaces.

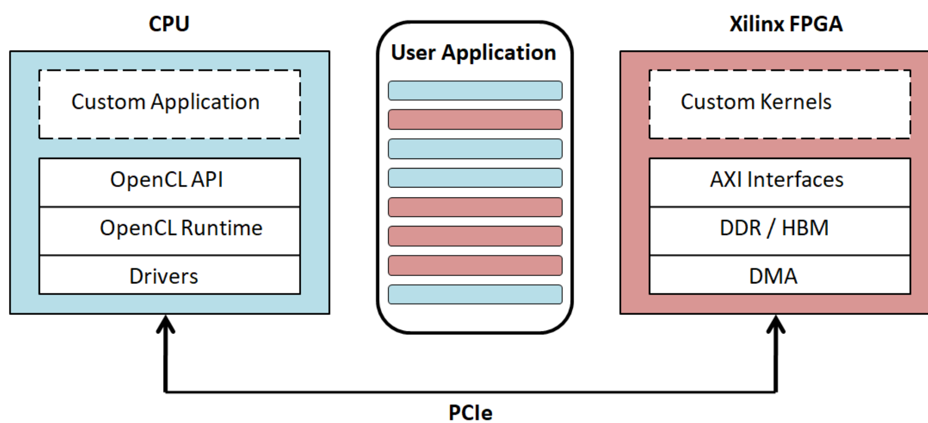

Figure 17. Software stack (left) and hardware components (right) communication through PCIe.

As it can be seen in the figure 17, the host code and the hardware kernels are provided by the developer of the accelerated application. Within this context, by using standard OpenCL and AXI interfaces, Alveo hides the complexity involved in building a platform based on an heterogeneous hardware architecture, and it lets the developers to focus on what matters most to them, getting the best in terms of performance out of their application. The application developer is not only designing the FPGA accelerated IPs but also creating customer code to be executed on the x86. The application execution stack is based on OpenCL, which is following a master-slave model which clearly separates the application code from the kernel logic. The host application submits work to FPGA kernels using standard OpenCL API, and the OpenCL runtime and drivers enable the communication between the host and the FPGA. The rationality behind the choice of

using OpenCL within the Alveo context is quite strong and it can be summarized in the following. OpenCL is being designed to be platform independent, which makes the code portable across CPUs, GPUs, and obviously FPGAs, which is again a huge opportunity when working within heterogeneous architecture. OpenCL is widely used, well-documented, and supported, and by using it, it is possible to dynamically wrap and load different kernels. The OpenCL runtime manages all of the communication between the host and the device, and takes away all the burden of the complexity of optimizing parallel computing application, delivering fast time to results.

To sum up, each Alveo u280 card includes three key components: a powerful FPGA for acceleration, high-bandwidth HBM2 memory banks, and high-bandwidth PCIe Gen3x16 connectivity to a host server. Between the Alveo card and the host, this link can transport around 16 gigabytes of data per second.

Alveo designs are split into a *shell and role* conceptual model to ensure that the PCIe link, system monitoring, and board health interfaces are always available to the host processor. External links, settings, timing, and other static functionality are all contained in the shell. The model's role component is filled with bespoke logic that implements the user's customized algorithms. This topology is reflected in figure 18.

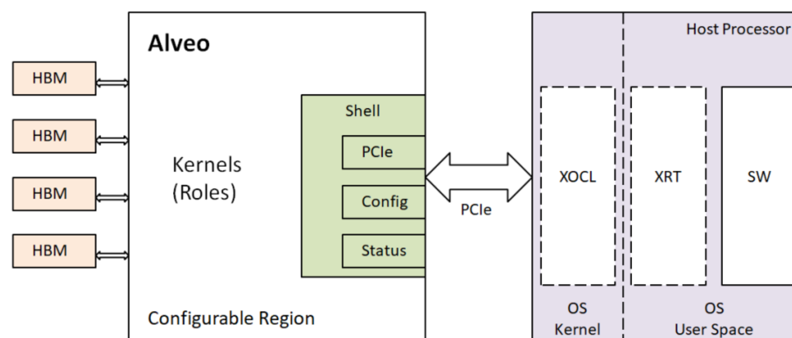

Figure 18. Alveo card shell and role topology.

Multiple super logic regions (SLRs) are separated into the Alveo FPGA, which aid in the building of very high-performance systems. Only the Alveo u280 has HBM memory banks, and the cards have several on-card DDR4 memories. These memories, which have a high bandwidth to and from the Alveo device, are collectively referred to as the *device global memory* in OpenCL. Each DDR4 memory bank has a 16-gigabyte capacity and runs at 2400 MHz DDR. This RAM has a lot of bandwidth, and modified kernels can easily saturate it if they want to. However, reading from or writing to this memory incurs a latency penalty, especially if the addresses accessed are not contiguous or the data beats are short.

4 GB HBM stacks are included in Alveo U280 High Bandwidth Memory (HBM) devices. The programmable logic connects with the HBM stacks via memory controllers using stacked silicon interconnect technology. The U280 has access to two 4 GB HBM stacks, each with 16 pseudo channels and direct 256 MB access. Below, there is a high-level

diagram of the two HBM stacks. There are 32 HBM AXI interfaces on the programmable logic. Through a built-in switch, HBM AXI interfaces can access any memory address in any of the 32 HBM PCs on any of the HBM stacks, giving them access to the entire 8 GB memory space. The programmable logic and HBM stacks' flexible connectivity allows for easy floorplanning and timing closure, as well as kernel implementation flexibility.

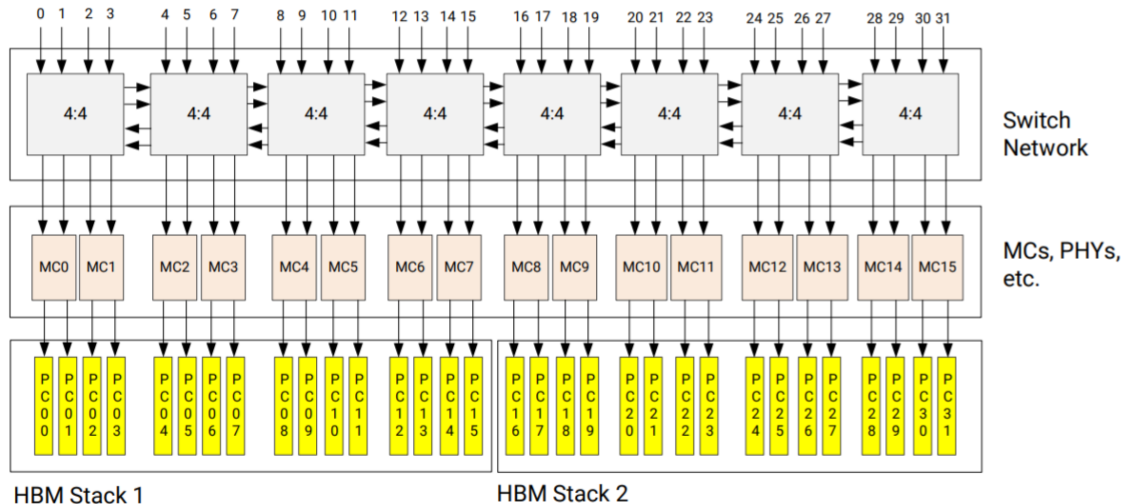

Figure 19. High-Level Diagram of Two HBM Stacks. [2]

3. Methodology

This section explains in detail the development of the project, the AXI interface design, as well as the connections between the interface and the Darwin co-processor, and its implementation making use of the High Bandwidth Memory Banks of the Alveo U280 FPGA.

3.1. AXI Interface

The Advanced eXtensible Interface (AXI) is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface primarily developed for on-chip communication.

With the AMBA3 specification, AXI was first presented in 2003. The AXI4, AXI4-Lite, and AXI4-Stream protocols were defined in 2010 by AMBA4, a new iteration of AMBA. Separate address/control and data phases, support for unaligned data accesses, burst-based transfers, separate and independent read and write channels, support for pending transactions, and more are all available with AXI.

AMBA AXI provides several optional signals that can be incorporated or excluded according on the design's individual requirements, making AXI an adaptable bus for a variety of applications.

While communication on an AXI bus is limited to a single master and slave, the specification includes precise descriptions and signals that allow the bus to be extended to topologies with multiple masters and slaves.

AXI defines a basic handshake mechanism consisting of two signals: xVALID and xREADY. The source sends the xVALID signal to the destination entity to alert it that the payload on the channel is valid and can be read from that clock cycle forward. Similarly, the receiving entity sends the xREADY signal to indicate that it is ready to accept data.

The data payload is deemed "transferred" when both the xVALID and xREADY signals are high in the same clock cycle, and the source can either send a fresh data payload by keeping xVALID high or stop the transmission by de-asserting xVALID. A single data transfer, such as a clock cycle in which both xVALID and xREADY are high, is referred to as a "beat."

For the control of these signals, there are two fundamental rules:

- A source must not assert xVALID until xREADY is high.
- A source must maintain a high xVALID until a handshake happens once claimed.

Both the source and the destination can regulate the flow of data using this handshake method, limiting the speed if necessary.

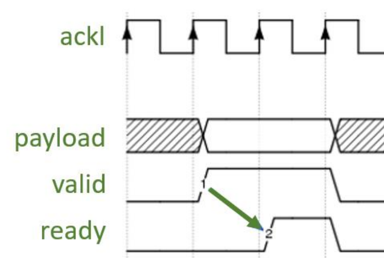

Figure 20. Basic handshake mechanism of the AXI protocol.

The destination entity in the example transaction shown in figure 20 waits for a high VALID before asserting its own READY.

Five channels are mentioned in the AXI specification:

- Read Address channel (AR)
- Read Data channel (R)
- Write Address channel (AW)
- Write Data channel (W)
- Write Response channel (B)

Aside from some basic ordering rules, each channel is self-contained, with its own set of xVALID/xREADY handshake signals.

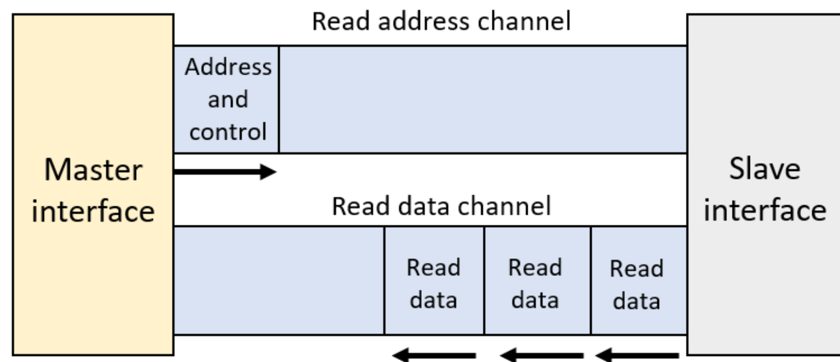

Figure 21. AXI Read Address and Read Data channels.

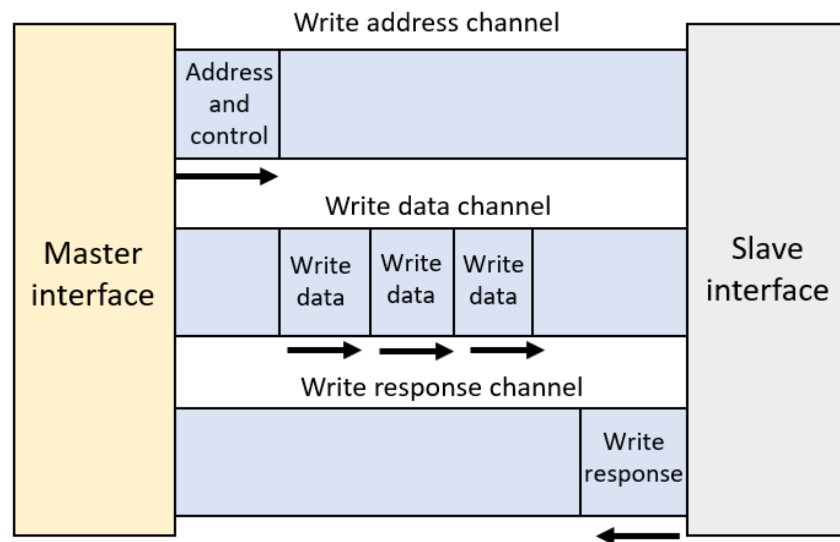

Figure 22. AXI Write Address, Write Data and Write Response channels.

AXI is a burst-based protocol, which means that a single request may result in numerous data transfers (or beats). This makes it handy in situations when a significant volume of data must be transferred from or to a specified pattern of addresses. The three types of bursts in AXI are FIXED, INCR, and WRAP, which are selectable by the signals ARBUSRTS (for reads) or AWBURST (for writes). Each beat inside the transmission has the same address in FIXED bursts. INCR bursts, on the other hand, have an address equal to the previous one plus the transfer size for each beat. WRAP bursts are similar to INCR bursts in that each transfer has an address equal to the previous one plus the transfer size; however, with WRAP bursts, the current beat's address is reset to the "Wrap boundary" if it crosses the "Higher Address boundary."

3.1.1. AXI Transactions

Reads

To initiate a read transaction, the master must give the start address on ARADDR, the burst type, either FIXED, INCR, or WRAP, on ARBUST (if present), and the burst duration on ARLEN on the Read address channel (if present).

If other auxiliary signals are provided, they are also utilized to define more particular transfers. The slave must send the data corresponding to the specified address(es) on RDATA and the status of each beat on RRESP, plus any additional optional signals, over the Read data channel after the standard ARVALID/ARREADY handshake.

Each beat of the slave response is signalled by an RVALID/RREADY handshake, and the slave must assert RLAST on the last transfer to indicate that no more beats will be sent without a new read request.

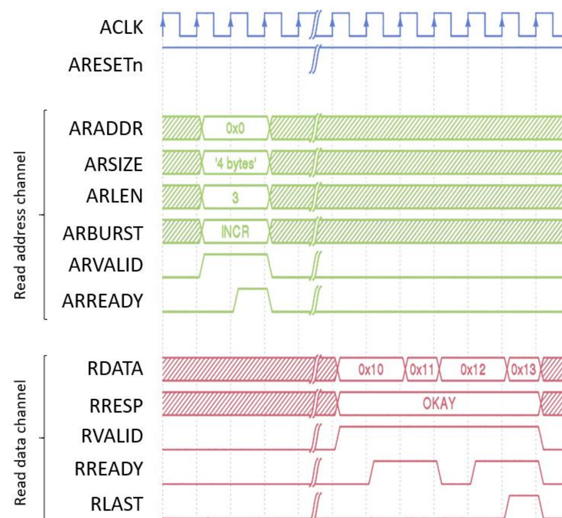

Figure 23. AXI Read transaction example.

Figure 23 shows how the master requests four beats (ARLEN+1) of four bytes each, starting at address 0x0, with INCR type. The slave returns 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8, and 0x13 for address 0xC. The figure only shows the most important signals.

Writes

To begin a write operation, the master must give both address and data information.

The start address must be provided on AWADDR, the burst type, either FIXED, INCR, or WRAP, on AWBURST (if present), the burst length on AWLEN (if present), and all the optional signals must be provided on the Write address channel, in a similar manner to a read operation.

A master must also send data on the Write data channel related to the given address(es); data on WDATA and the "strobe" bits on WSTRB, which conditionally label particular WDATA bytes as "valid" or "invalid."

The master must assert WLAST on the last data word, just like in the read path. After both transactions are complete, the slave must report the status of the write to the master over the Write response channel, returning the result via the BRESP signal.

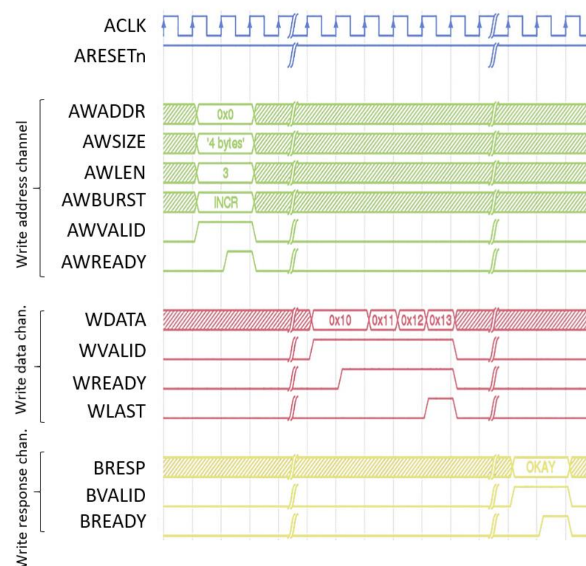

Figure 24. AXI Write transaction example.

In the example of figure 24, the master writes 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8, and 0x13 for address 0xC in four beats (AWLEN+1) each starting from address 0x0 with INCR type. For the entire transaction, the slave returns 'OKAY' as a write response. The figure only shows the most important signals.

3.1.2. AXI4-Lite

AXI4-Lite is a subset of the AXI4 protocol that offers a register-like structure with fewer features and a lower level of complexity. The fact that all bursts are composed of only one beat and that all data accesses use the full data bus width, which can be either 32 or 64 bits, are notable contrasts.

AXI4-Lite omits several AXI4 signals while adhering to the AXI4 specification for the rest. Its transactions are fully compatible with AXI4 devices since they are a subset of AXI4, allowing for interoperability between AXI4-Lite masters and AXI4 slaves without the need for additional conversion logic.

3.1.3. AXI interface implementation

The GACT array accelerators handle the compute-intensive *align* routine in GACT, with the rest of the algorithm implemented in software. The arrays are originally implemented on an Intel FPGA. In this work, they have been implemented using the Alveo Card U280, exploiting HBM to increase its performance.

The kernel has been written in RTL, and the communication between the host and the kernel occurs across an AXI bus. The main data processed by the kernel, which is in a large volume, is transferred through the global memory banks on the FPGA board, making use of 6 HBM banks. The kernel accesses the data from those global memory banks, in burst, and after it finishes the computation, the resulting data is transferred back to the host machine through the HBM memory banks.

The AXI4 master interfaces for global memory access are a total of six, which all have 64-bit addresses, which are the query sequence and the reference sequences with their respective addresses, and the direction read address. Each partition in the global

memory becomes a kernel argument, and the memory offset for each partition is set by a control register programmed via the AXI4-Lite slave interface.

With respect to the scalar arguments, which are directly written to the kernel through the AXI4-Lite slave interface, there are a total of 11, which are the input register for the Darwin GACT logic. These inputs are control variables directly loaded from the host machine and they do not use global memory banks.

The RTL Darwin kernel consists of a top-level Verilog design which contains a control register and the Darwin sub-modules with a read and write module for each AXI4 master. The following figure illustrates the top-level design configured with six AXI4-master interfaces.

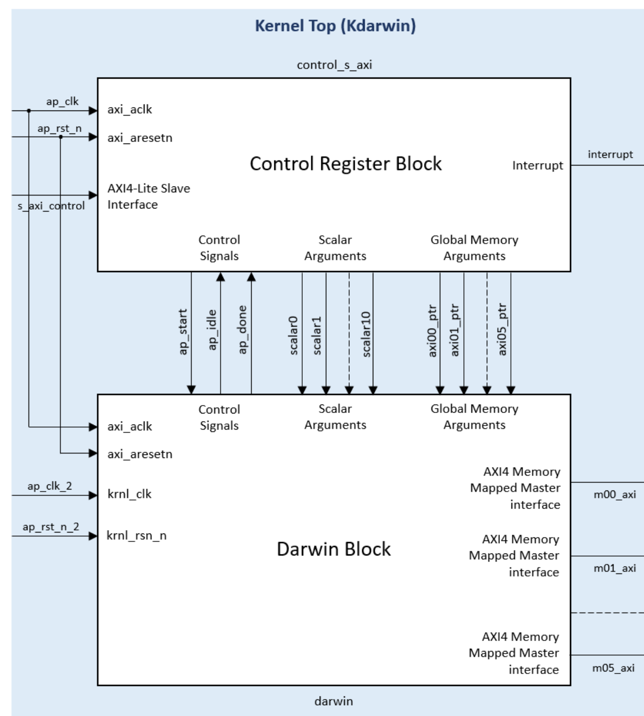

Figure 25. Top-level Darwin kernel structure.

The Darwin block, shown in the following figure, consists of the GACT Darwin logic, six AXI4 read masters, and six AXI4 write masters. Each master reads 16 KB of data, performs the necessary operations in the Darwin logic block, and then writes out the Darwin outputs back in place. Each master AXI uses a different HBM memory bank, from HBM[0] to HBM[5].

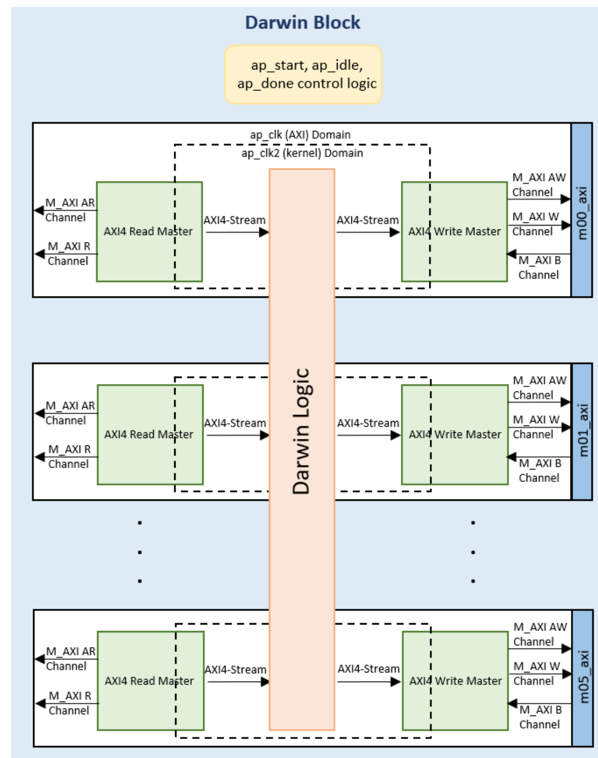

Figure 26. Darwin kernel RTL block structure.

The design environment used for this project has been Vivado Design Suite and Vitis Unified Software Platform, and the HDL design has been written down in Verilog and SystemVerilog.

Vivado Design Suite is a software suite for synthesis and analysis of HDL designs, produced by Xilinx, which replaces Xilinx ISE and adds functionality for system on a chip development and high-level synthesis. Vivado is a complete rewriting and rethinking of the entire design process (compared to ISE). ISIM, an in-built logic simulator, is included.

On heterogeneous Xilinx platforms such as FPGAs, SoCs, and Versal ACAPs, the Vitis Unified Software Platform facilitates the development of embedded software and accelerated applications. It offers a single programming model for Edge, Cloud, and Hybrid computing applications.

External memory interfaces, custom input/output interfaces, and software runtime are all part of the Vitis target platform, which provides the fundamental hardware and software architecture as well as the application context for Xilinx platforms.

The Vitis target platform automatically configures the PCIe interfaces that connect and manage communication between the FPGA accelerators and the x86 Application code for Xilinx accelerator cards on-premise or in the cloud.

In the current project, the interfaces of the original Darwin co-processor have been adapted to the Xilinx boards, specifically for the Alveo U280 Accelerator Card, in order to make use of the High Bandwidth Memory banks (HBM), since it is a much faster RAM, and the fact that the original Darwin co-processor was memory bound can be exploited.

The following table shows the type of AXI interface for each GACT input of the Darwin co-processor.

Table 2. AXI configurations used for each GACT input in the RTL kernel.

Darwin inputs	AXI configurations
in_params	AXI master
set_params	AXI-Lite
query_in	AXI master
ref_in	AXI master
ref_addr_in	AXI master
query_addr_in	AXI master
ref_wr_en	AXI-Lite
query_wr_en	AXI-Lite
max_tb_steps	AXI-Lite
ref_len	AXI-Lite
query_len	AXI-Lite
score_threshold	AXI-Lite
align_fields	AXI-Lite
start	AXI-Lite
clear_done	AXI-Lite
dir_rd_addr	AXI master
req_id_in	AXI-Lite

3.2. Block Design

The RTL Kernel has been created using the Vitis RTL Kernel wizard. This allows RTL code to be used in a Vitis design. The kernel has been written in Verilog and SystemVerilog. The project has been developed using Vivado Design Suite. After the simulation verification of the design, the generated Vivado project has been packed into an xo kernel file.

The eleven scalar kernel input arguments (AXI-Lite configuration) are all *uint* type (32 bit long), and the six arguments passed through Global Memory are of width 64 bytes, or which is the same, 512 bits.

The function prototype and register map for the kernel are shown in the following figure. Note that the control register and the scalar operands are accessed via the S_AXI_CONTROL interface. The control register is at offset 0x0 and the scalar operands start at offset 0x10.

VLNV: xilinx.com:kernel:KDarwin:1.0

Target platform: xilinx:u280:xdma:201920.3

Function prototype: **void KDarwin**(**const uint** set_params, **const uint** ref_wr_en, **const uint** query_wr_en, **const uint** max_tb_steps, **const uint** ref_len, **const uint** query_len, **const uint** score_threshold, **const uint** align_fields, **const uint** start, **const uint** clear_done, **const uint** req_id_in, **global void** *in_params, **global void** *query_in, **global void** *ref_in, **global void** *ref_addr_in, **global void** *query_addr_in, **global void** *dir_rd_addr);

Register map:

ID	Name	Offset	Type (bits)	Interface
N/A	Control	0x000	N/A	S AXI CONTROL
0	set_params	0x010	uint (32)	S_AXI_CONTROL
1	ref_wr_en	0x018	uint (32)	S_AXI_CONTROL
2	query_wr_en	0x020	uint (32)	S_AXI_CONTROL
3	max_tb_steps	0x028	uint (32)	S_AXI_CONTROL
4	ref_len	0x030	uint (32)	S_AXI_CONTROL
5	query_len	0x038	uint (32)	S_AXI_CONTROL
6	score_threshold	0x040	uint (32)	S_AXI_CONTROL
7	align_fields	0x048	uint (32)	S_AXI_CONTROL
8	start	0x050	uint (32)	S_AXI_CONTROL
9	clear_done	0x058	uint (32)	S_AXI_CONTROL
10	req_id_in	0x060	uint (32)	S_AXI_CONTROL
11	in_params	0x068	generic pointer (64)	m00_axi
12	query_in	0x074	generic pointer (64)	m01_axi
13	ref_in	0x080	generic pointer (64)	m02_axi
14	ref_addr_in	0x08c	generic pointer (64)	m03_axi
15	query_addr_in	0x098	generic pointer (64)	m04_axi
16	dir_rd_addr	0x0a4	generic pointer (64)	m05_axi

Figure 27. Function prototype and register map for the Darwin kernel.

There is one module to handle the control signals (ap_start, ap_done and ap_idle) and six master AXI channels to read source operands from, and write the result to HBM, each with its read and write instances.

The two top-level blocks, Darwin block and Control Register block, are shown below.

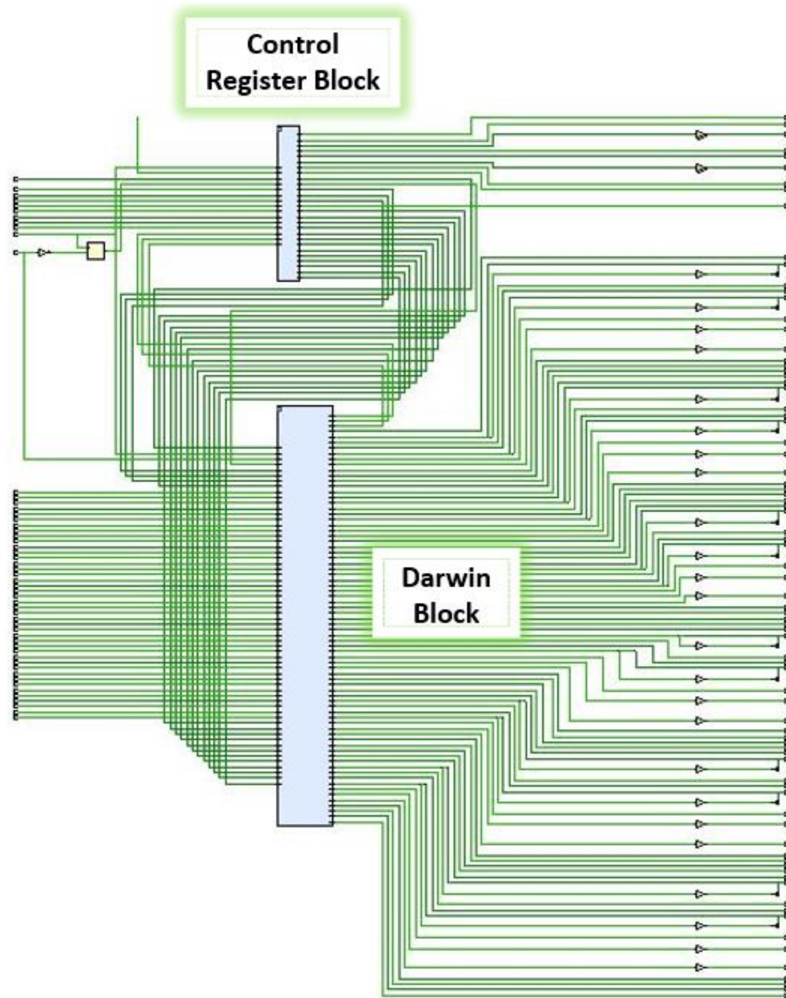

Figure 28. Top-level blocks of the design (control register and Darwin).

Inside the Darwin Block of Figure 28, it can be observed the six hierarchical Master AXI read and write blocks, as well as the Darwin Logic block, as illustrated in Figure 26.

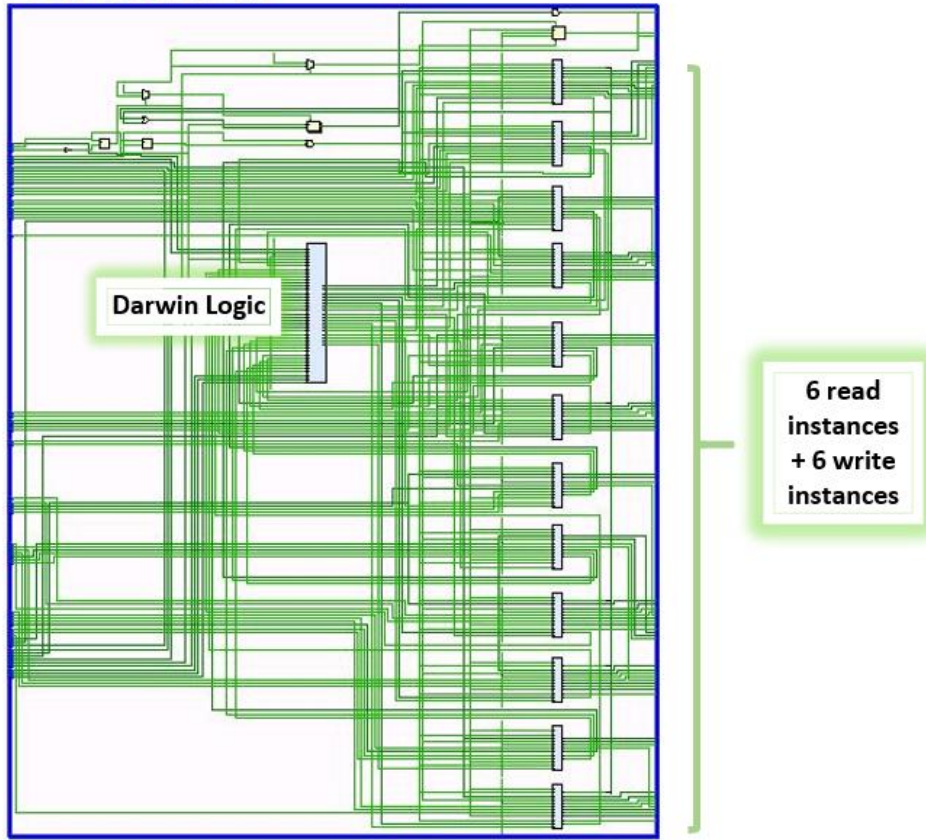

Figure 29. Darwin Block design.

As Figure 29 shows, there are six AXI master interface blocks, each one consisting of a Read Master and a Write Master.

3.3. Simulations

For simulation purposes, and to make sure that all the signals are mapped correctly, two simple strings have been used, shown in Figure 30, which result in a tile score (and also maximum score) of 3.

		A	T	G	C	T
	0	0	0	0	0	0
A	0	1	0	0	0	0
G	0	0	0	1	0	0
C	0	0	0	0	2	0
T	0	0	0	0	0	3

Figure 30. Query and reference sequences used for simulation.

The string *"A T G C T"* is the query sequence *Q*, while the *"A G C T"* is the reference sequence *R*.

Darwin has an input *in_params* that consist of the score matrix, gap_open and gap_extend parameters. This input is an array of 1's and -1's. In the simulation example PE_WIDTH = 10, so the size of *in_params* result in 10x12 bits, this is 120 bits.

The following figure shows a simulation result for the input sequences of Figure 30. As it can be seen, the tile score corresponds to the maximum score since there is only one single tile to compute, and it is 3, as expected.

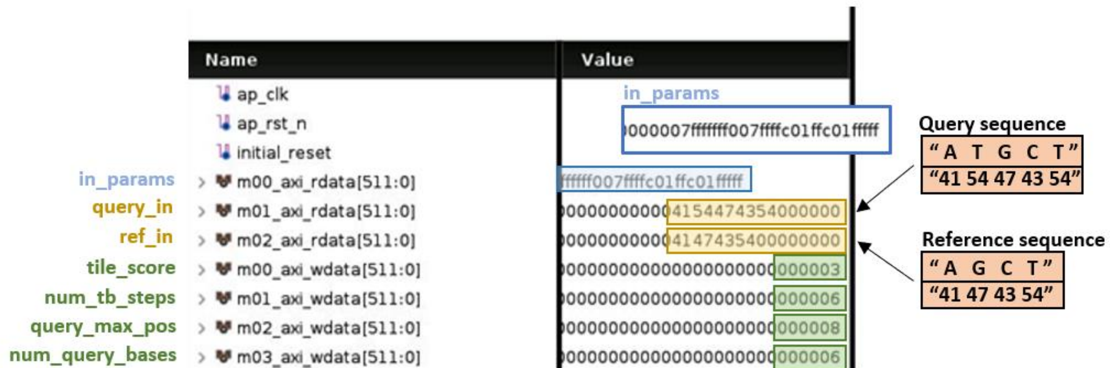

Figure 31. Simulation results for the query "ATGCT" and reference sequence "AGCT".

The following simulation shows the AXI signals for the AXI Master 0, where there can be seen some of the ones explained in section 3.1.1.

Figure 32. AXI Master 0 interface signals in Vivado Simulation.

Figure 33 shows the simulation of some Darwin Logic block signals as well as the corresponding master AXI interface values. As it can be seen, there are two state values, the upper one, in yellow, that is the state of the Darwin Logic FSM, and the one marked in

blue, corresponding to the state of the Smith Waterman Array FSM. When the array processing is done, and the Smith Waterman Array FSM state is set as "Done", the tile score updates its value. This is written by the master AXI 0 write instance to the HBM. As an example, and marked in the figure, it can be observed the query input and the reference input sequences, both corresponding to the data read by the master AXI 1 and 2 interfaces, respectively, from the HBM.

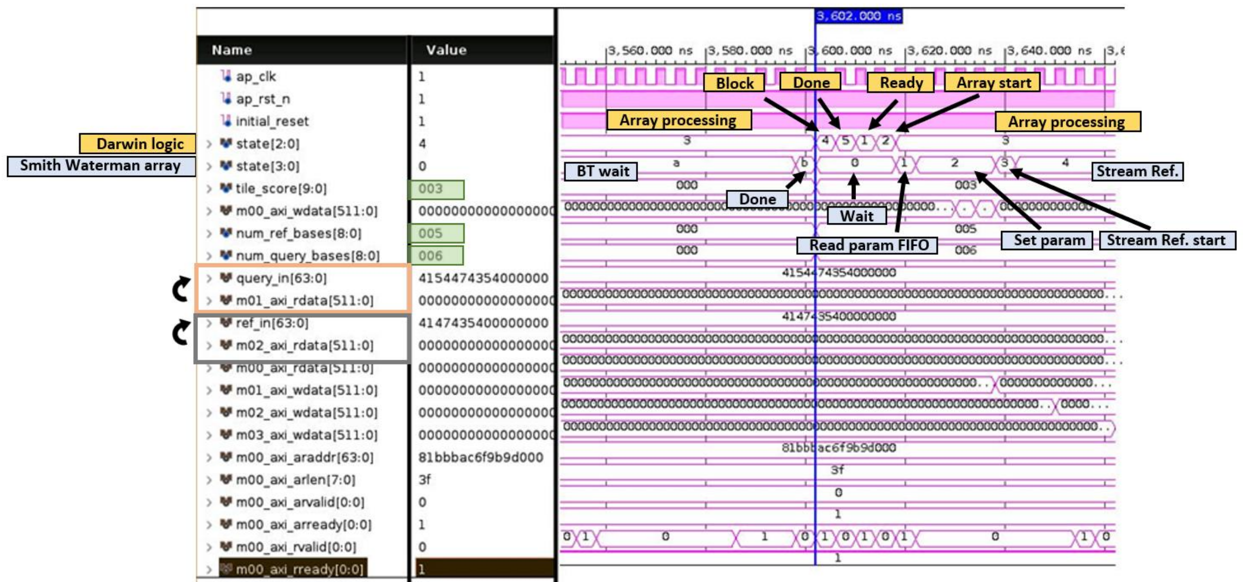

Figure 33. Darwin Logic simulation results for Q = "ATGCT" and R = "AGCT".

The testbench is formed by six slave instances, one for each AXI master.

3.4. High Bandwidth Memory (HBM)

Although the field-programmable gate array (FPGA) is recognized to provide a high-performance and energy-efficient solution for many applications, memory-bound applications, such as the Darwin co-processor, are generally thought to be less competitive.

However, with the recent introduction of High Bandwidth Memory 2 (HBM2) FPGA boards, future FPGAs may be able to compete with GPUs in memory-bound applications. The Alveo U280 from Xilinx has a potential bandwidth of roughly 400 GB/s (two HBM2 DRAMs), which is comparable to the Titan V GPU from Nvidia (650 GB/s, three HBM2 DRAMs). This HBM-based FPGA technology has the potential to enable a larger range of applications to benefit from FPGA acceleration because to its high memory bandwidth.

HBM2 increases the maximum I/O data rate from 1 Gbps (HBM1) to 2 Gbps. The use of two pseudo channels (PCs) per physical channel to mask latency helps to achieve this. Each stack has sixteen PCs that can be accessed independently.

Massive memory bandwidth is provided by HBMs. HBM provides up to 425 GB/s memory bandwidth on the target FPGA board Alveo U280, which is an order of magnitude larger

than using two standard DDR4 channels on the same board. Although this is only half of what state-of-the-art GPUs can do, it is a big step forward for FPGAs.

High Bandwidth relies heavily on the Address Mapping Policy. When running a typical memory access pattern (i.e., sequential traversal) on HBM, different address mapping policies result in an order of magnitude difference in throughput, demonstrating the significance of matching the address mapping policy to the application.

In the FPGA, Xilinx combines two HBM stacks and an HBM controller, as shown in Figure 34. Each HBM stack is divided into eight memory channels, each of which is further subdivided into two 64-bit pseudo channels. As illustrated in figure 35, a pseudo channel can only access its associated HBM channel, which has its own address region of memory. There are 16 memory channels, 32 pseudo channels, and 32 HBM channels in the Xilinx HBM subsystem. There are 32 AXI channels that communicate with the user logic on top of the 16 memory channels. To provide a proven standardized interface to the FPGA programmer, each AXI channel follows the standard AXI4 protocol. Because each AXI channel is linked to an HBM channel (or pseudo channel), each AXI channel can only access its own memory.

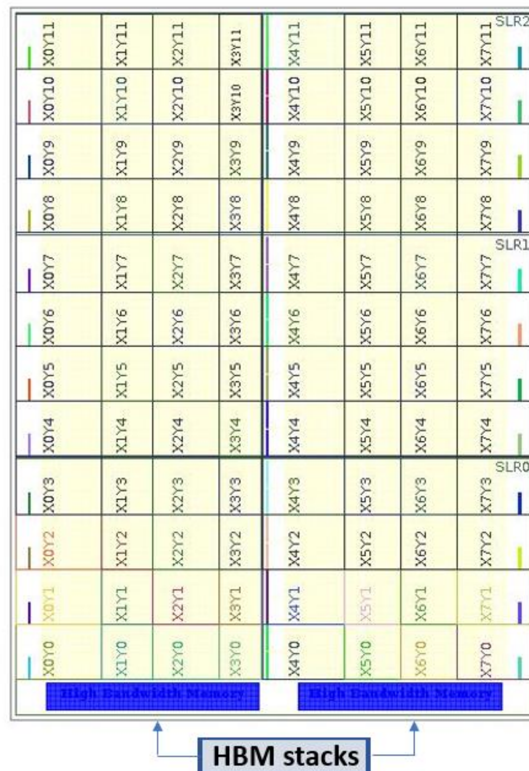

Figure 34. Alveo U280 board with the two HBM stacks.

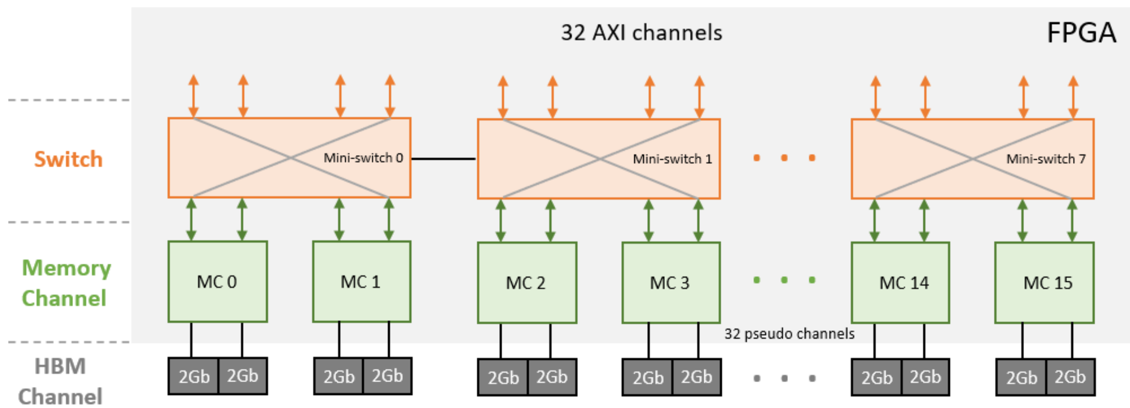

Figure 35. Architecture of Alveo HBM subsystem.

The following figure shows the platform diagram, obtained through the Vitis Analyzer, as it can be observed there are 32 HBM memory banks that communicate via PCIe with the host processor and FPGA Alveo U280.

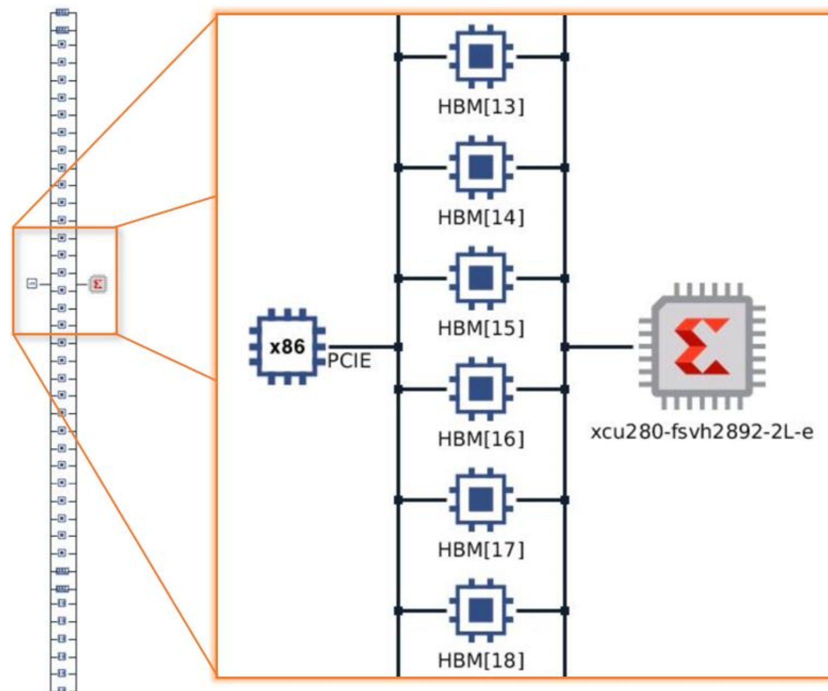

Figure 36. Platform diagram in Vitis Analyzer tool.

The information of the kernel bitstream generation is depicted in Figure 37. The number of kernels is set to 1, as marked in orange, so only a single compute unit is created, this is a single kernel. Six arguments are mapped for the six different HBM banks used, one for each AXI master, these banks are from HBM[0] to HBM[5].

```
Build Console [darwin_kernel, Hardware]
INFO: [CFGEN 83-0] Kernel Specs:
INFO: [CFGEN 83-0]   kernel: KDarwin, num: 1  {KDarwin_1}
INFO: [CFGEN 83-0] Port Specs:
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: in_params, sptag: HBM[0]
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: query_in, sptag: HBM[1]
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: ref_in, sptag: HBM[2]
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: ref_addr_in, sptag: HBM[3]
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: query_addr_in, sptag: HBM[4]
INFO: [CFGEN 83-0]   kernel: KDarwin_1, k_port: dir_rd_addr, sptag: HBM[5]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.in_params to HBM[0] for directive KDarwin_1.in_params:HBM[0]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.query_in to HBM[1] for directive KDarwin_1.query_in:HBM[1]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.ref_in to HBM[2] for directive KDarwin_1.ref_in:HBM[2]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.ref_addr_in to HBM[3] for directive KDarwin_1.ref_addr_in:HBM[3]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.query_addr_in to HBM[4] for directive KDarwin_1.query_addr_in:HBM[4]
INFO: [CFGEN 83-2228] Creating mapping for argument KDarwin_1.dir_rd_addr to HBM[5] for directive KDarwin_1.dir_rd_addr:HBM[5]
INFO: [SYSTEM LINK 82-37] [12:19:59] cfgen finished successfully
```

Figure 37. Information about the kernel bitstream generation, provided by Vitis.

Figure 38 shows the system diagram, obtained through the Vitis Analyzer tool, where there can be observed the host processor with its PCIe communication with the FPGA, the six HBM memory banks used, and its connexions with the kernel (KDarwin_1). The rest of the kernel arguments are AXI-Lite parameters, and they are not passed through global memory.

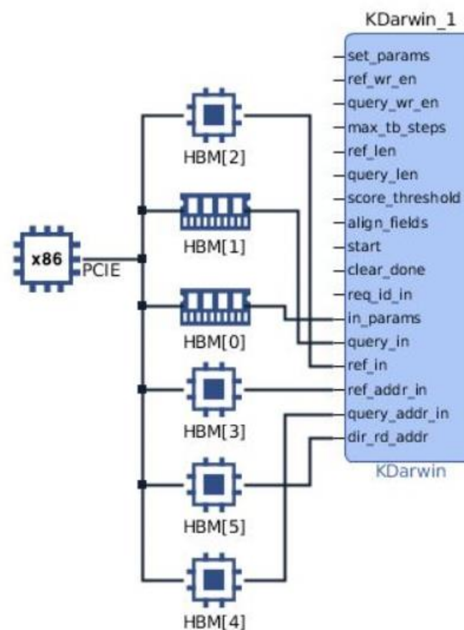

Figure 38. Darwin Kernel system diagram obtained through Vitis HLS.

3.5. Host Processor

The host code has been described using C++ and it has been developed using the OpenCL.

The host code initializes the inputs and indicates to the FPGA that it has 32 HBM memory banks of data (in the Alveo U280). A five-vector input are created, of the same size that they have in the actual architecture. They are created as aligned allocators, because in this way is easier for the host part to send data if it is already aligned. In this way, the host does not have to reorganize the data for the FPGA beforehand to send it aligned.

The host code contains the kernel name, named KDarwin, which is the one of the top module in the kernel. Then a vector of kernels has been created in order to have the possibility to implement a multicore architecture. Instead of looking for just one kernel, it will look for six instances of the same kernel, and then they will be mapped in the FPGA.

Then, the host programs the FPGA, finding the bitstream file, which is passed as the first argument, in order to create a context, and a command queue to communicate with the FPGA, and program the device (the Alveo U280).

The number of kernels is set to one because is designed for only a compute unit, but if multiple instances are created, it iterates them to the FPGA, since the vector of kernels has been created in the host file.

In the host code there is also the mapping of the HBM memory banks, from bank 0 to bank 5, in order. Since the architecture is done to be able to be multicore, with only 1 Compute Unit it only runs for a single iteration, but with more Compute Units it can be ran for more iterations. In this way it creates the AXI master interface connections with the HBM memory banks.

Finally, it sets the inputs to the FPGA, setting the arguments to the Alveo U280, checking that they are correct, and executing the kernel, enqueuing the commands to the FPGA.

4. Results

The final design has been implemented in the Alveo U280 Accelerator Card, exploiting HBM to increase the performance of the co-processor.

The following table summarizes the resource utilization in the target FPGA, where there can be seen the resources used by the two main blocks, the control register and the Darwin block.

Table 3. Resource utilization of the final design implemented in the Alveo U280.

	CLB LUTs (1303680)	CLB Registers (2607360)	CARRY8 (162960)	F7 Muxes (651840)	F8 Muxes (325920)	CLB (162960)	LUT as Logic (1303680)	LUT as Memory (600960)	Block RAM Tile (2016)
KDarwin	8057	17414	244	101	5	1920	4353	3704	43
Control Register Block	683	791	0	42	0	172	683	0	0
Darwin Block	7374	16622	244	59	5	1832	3670	3704	43

The timing design summary is depicted in the following table, obtaining a worst negative slack of 0,022 ns.

Table 4. Design timing summary.

Design Timing Summary					
WNS (ns)	TNS (ns)	TNS Total Endpoints	WHS (ns)	THS (ns)	THS Total Endpoints
0,022	0,000	554885	0,001	0,000	552084

The implementation of the design resulted in 145.666.450 Cell Updates Per Second (CUPS) per tile, with a tile size set to the maximum which is 512 (T=512), this is 146 Mega CUPS per tile. Cell Update Per Second (CUPS) is a standard measurement unit widely used in these kinds of algorithms.

In the Darwin co-processor, Turakhia et al. [1] synthesized 40 GACT arrays of 32 PEs on the Arria 10 FPGA. GACT arrays on FPGA provided a peak throughput of 1.3 Milion tiles per second with the tile size set at 320 (T=320), thus having a sequence length of 320, as shown in Figure 39.

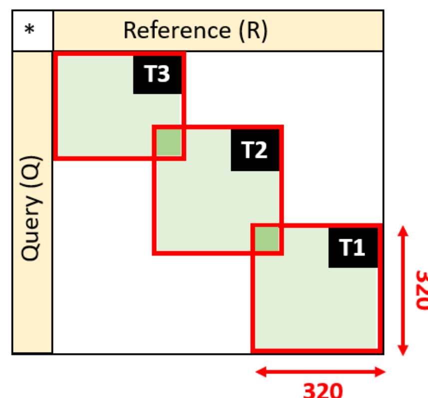

Figure 39. GACT array for tile size T = 320.

The actual design is only synthesized for 1 GACT array on the Alveo U280 FPGA. In order to compare the results with the original Darwin co-processor [1], the result of 1,3 M tiles per second for 40 GACT arrays has to be divided by 40, thus obtaining 32500 tiles per second for 1 GACT array.

Multiplying 32500 tiles times 320 x 320, the tile size, Cell Updates Per Second are obtained as a result, since each tile has 320x320 cells:

$$32500\ tiles\ x\ (320\ x\ 320)\ cells\ = 3{,}328\ GCUPs \tag{5}$$

Thus, the original Darwin co-processor [1], with 32 Processing Elements, and the tile size set at *T*=320, results in a performance of 3,328 Giga cells update per second.

The actual kernel design, implemented on the Alveo U280, of the current project has been tested for 4 PEs and *T*=320, obtaining a result of 140 µs to compute a single tile.

To obtain the actual cell updates per second, the following calculation has been applied:

$$\frac{(320\ x\ 320)\ cells}{140\ \mu s} = 0{,}73\ GCUPs \tag{6}$$

Obtaining a result of 0,73 GCUPs for 4 Processing Elements. Since the original Darwin implementation has been made with 32 PEs, in order to be able to compare both performances, the result has to be multiplied by 8, thus giving a final result of 5,851 GCUPs.

The speedup obtained results in **1,75X**, from the original Darwin giving 3,328 GCUPs to the accelerated one giving **5,851 GCUPs**.

5. Budget

In this section, a study on the cost of the project is developed, taking into account both the material used as well as the hours required to carry it out.

The duration of the project has been a total of 16 weeks. Table 5 shows the temporal analysis of each task performed during this period, with the hours devoted to each of them.

Table 5. Temporal analysis of the project.

Task	Hours
Scope definition	10
Contextualization and bibliography	30
Development environment configuration (hardware)	3
Development environment configuration (software)	5
Documentation study	90
Darwin co-processor simulation	10
AXI interface design	40
Darwin block and interface connection design	190
Testbench design	40
RTL simulations	60
HBM configuration	55
Bitstream generation	60
Host code development in C++	50
FPGA implementation	40
Final documentation presentation	90
TOTAL	**773**

This total of 773 hours translates into a total of 6,9 hours per day.

The resources used during the project development are described below, and they have been available for their usage 24 hours a day, 7 days a week.

- **Hardware:**
  - Personal Laptop
  - Computer
  - Alveo U280 Accelerator Card
- **Software:**
  - OpenVPN
  - Vivado Design Suite
  - Vitis Unified Software Platform
  - Open Office
  - Slack
- **RRHH:**
  - Project director
  - Analyst
  - Programmer
  - Testing programmer

Therefore, a budget will be made based on the hours of work involved in the completion of each of the stages.

Table 6. Breakdown of the project budget.

Direct costs						
Resource	**Quantity**	**Price per unit**	**Time**	**% Amortization**	**Total**	**Observations**
Project director	-	25 €/h	20 h	-	**500 €**	-
Analyst	-	22 €/h	8 h	-	**176 €**	-
Programmer	-	19 €/h	495 h	-	**9405 €**	-
Testing programmer	-	18 €/h	250 h	-	**4500 €**	-
Computer	1	1450 €	-	8,33 %	**120,83 €**	Shelf life 4 years, 4

						months of use
Personal Laptop	1	1050 €	-	8,33 %	**87,50 €**	Shelf life 4 years, 4 months of use
Alveo U280 Accelerator Card	1	7590	-	4,76 %	**361,28 €**	Shelf life 7 years, 4 months of use
OpenVPN	-	0 €	-	-	**0 €**	-
Vivado Design Suite	-	0 €	-	-	**0 €**	-
Vitis Unified Software Platform	-	0 €	-	-	**0 €**	-
Open Office	-	0 €	-	-	**0 €**	-
Slack	-	0 €	-	-	**0 €**	-
Indirect costs						
Resource	**Quantity of power**	**Price per unit**	**Time**	**% Amortization**	**Total**	**Observations**
Computer	0,40 kW/h	0,149 €/kWh	600 h	-	**35,76 €**	400 W of power
Personal Laptop	0,25 kW/h		700 h	-	**26,08 €**	250 W of power
Alveo U280 Accelerator Card	0,225 kW/h		150 h	-	**5,03 €**	225 W of power
Total budget	**15217,48 €**					

The total project budget is quite high due, mainly, to the hours of work required by the programmers.

6. Conclusions and future development

This work presents a genomic co-processor for long read alignment, the HPC FPGA implementation of the GACT alignment algorithm.

Detailed results and analyses show significant performance acceleration exploiting High Bandwidth Memory. This project demonstrated runtime improvements of up to **1,75x** using the Xilinx Alveo U280, compared with the original Intel FPGA implementation.

The implementation has been done mapping six HBM memory banks of the target FPGA to the six AXI master interfaces of the kernel design, being just one Compute Unit (CU) in the design, so a single kernel. In order to increase even more the performance, future work can focus on implementing more Compute Units inside the FPGA, since the host file is already prepared for it. With more Compute Units the improvement can be even better than 1,75x with respect to the original GACT implementation, since more HBM memory banks can be exploited.

As future work, the seeding construction can be integrated on Hardware, which is something that Darwin does not perform yet.

The algorithms can be integrated within software used in the state of the art (Minimap, BWA-MEM, etc.) and within Darwin itself.

Bibliography

[1] Y. Turakhia, G.Bejerano, and W. J. Dally. "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly." *ACM SIGPLAN Notices,* vol. 53. ACM, 2018. [Consultation: 22/02/2021]

[2] Xilinx, 2020. *Alveo U280 Data Center Accelerator Card User Guide*. https://www.xilinx.com/support/documentation/boards_and_kits/acceleratorcards/ug1314-u280-reconfig-accel.pdf. [Consultation: 30/02/2021]

[3] K.-M. Chao, W. R. Pearson, and W. Miller. "Aligning two sequences within a specified diagonal band". *Computer applications in the biosciences: CABIOS*, 8(5):481–487, 1992. [Consultation: 03/03/2021]

[4] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. "A greedy algorithm for aligning dna sequences". *Journal of Computational biology*, 7(1-2):203–214, 2000. [Consultation: 07/03/2021]

[5] G. Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". *Journal of the ACM (JACM),* 46(3):395–415, 1999. [Consultation: 07/03/2021]

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. "Basic local alignment search tool". *Journal of molecular biology*, 1990. [Consultation: 07/03/2021]

[7] I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan. "Fast and sensitive mapping of nanopore sequencing reads with graphmap". *Nature communications*, 7, 2016. [Consultation: 07/03/2021]

[8] M. J. Chaisson and G. Tesler. "Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory". *BMC bioinformatics*, 13(1):238, 2012. [Consultation: 15/03/2021]

[9] G. Myers. "Efficient local alignment discovery amongst noisy long reads". *International Workshop on Algorithms in Bioinformatics*, pages 52–67. Springer, 2014. [Consultation: 15/03/2021]

[10] H. Li. "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem". *arXiv preprint arXiv*:1303.3997, 2013. [Consultation: 15/03/2021]

[11] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, and A. M. Phillippy. "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation". *bioRxiv*, page 071282, 2016. [Consultation: 21/03/2021]

[12] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". *Nature biotechnology*, 33(6):623–630, 2015. [Consultation: 22/03/2021]

[13] J. Buhler. "Efficient large-scale sequence comparison by locality sensitive hashing". *Bioinformatics*, 17(5):419–428, 2001. [Consultation: 25/03/2021]

[14] TimeLogic Corporation. URL http://www.timelogic.com. [Consultation: 25/03/2021]

[15] Y. Turakhia, G.Bejerano, and W. J. Dally. *Darwin Github*. https://github.com/yatisht/darwin [Consultation: 22/02/2021]

[16] Xilinx, 2021. *AXI High Bandwidth Memory Controller v1.0* https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf [Consultation: 05/04/2021]

[17] Y. Choi, Y. Chi, J. Want, L. Guo, and J. Cong, "When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization". *arXiv:2010.06075v1 [cs.AR],* October 2020. [Consultation: 10/04/2021]

[18] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking High Bandwidth Memory on FPGAs". *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020. [Consultation: 10/04/2021]

[19] Xilinx. 2020. *Vitis Unified Software Platform*. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html [Consultation: 25/04/2021]

[20] ARM. 2011. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite.* www.arm.com [Consultation: 10/04/2021]

[21] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim, "HBM (High Bandwidth Memory) DRAM technology and architecture". *Proc. IEEE Int, Memory Workshop*, 1-4, 2017. [Consultation: 01/05/2021]

[22] A. Zeni, G. W. Di Donato and F. Peverelli. *PALADIN Github*. https://github.com/albertozeni/XDropXOHW-Public [Consultation: 01/06/2021]

Appendices

The RTL kernel design can be found attached. A summary of the files included in the design are provided below:

- **KDarwin.v:** Kernel top module written in Verilog
- **KDarwin_block.sv:** Darwin block written in SystemVerilog
- **KDarwin_control_s_axi.v:** Control Register block written in Verilog
- **KDarwin_logic.v:** Darwin logic written in Verilog
- **KDarwin_axi_read_master.sv:** AXI4 Master Read module written in SystemVerilog
- **KDarwin_axi_write_master.sv:** AXI4 Master Write module written in SystemVerilog
- **KDarwin_counter.sv:** Counter module written in SystemVerilog
- **Ascii2Nt.v:** Darwin logic submodule written in Verilog
- **BRAM.v:** Darwin logic submodule written in Verilog
- **BRAM_QR.v:** Darwin logic submodule written in Verilog
- **BTLogic.v:** Darwin logic submodule written in Verilog
- **DP_BRAM.v:** Darwin logic submodule written in Verilog
- **FIFOWithCount.v:** Darwin logic submodule written in Verilog
- **mux_1OfN.v:** Darwin logic submodule written in Verilog
- **Nt2Param.v:** Darwin logic submodule written in Verilog
- **SmithWatermanArray.v:** Darwin logic submodule written in Verilog
- **SmithWatermanPE.v:** Darwin logic submodule written in Verilog
- **KDarwin_tb.sv:** Kernel testbench written in SystemVerilog
- **binary_container_1.xclbin:** Bitstream file