Hardware Accelerator for Genome Sequence Alignment

Shashank Nag EE19B118[1], **Sai Gautham Ravipati EE19B053**[1], **Vishnu Varma V EE19B059**[1]
[1]Department of Electrical Engineering, Indian Institute of Technology, Madras

1 Introduction

The genome sequence is a sequence of base pairs that constitute the DNA strands. The sequencers that read the DNA strands generate short-read sequences with errors, which have to be subsequently aligned to a reference genome. The aim is to identify the exact original sequence, which is important from a biological perspective. One popular algorithm for performing this alignment is the Smith-Waterman Algorithm, which has a high compute time complexity of $\mathcal{O}$(mn) - m & n being sequence lengths. It aligns the read to the target by breaking them into subsequences, and these aligned subsequences are extended to obtain the complete aligned sequence. The algorithm considers the effects of errors such as mismatch, insertion, and deletion. When such errors occur, they have to be penalised and when the corresponding base pairs of the sequences match, they have to be rewarded. Based on this scheme, a scoring matrix is constructed and computed where the alignment scores across both the sequences are recursively calculated based on matches and errors and updated in the matrix. Once the scoring is completed, the traceback phase is started to find the optimal alignment. Starting from the highest scoring cell in the matrix, we trace backwards to the cell which resulted in the highest score among its adjacent cells, till we hit a null score. Based on the direction of the traceback, the alignment would have gaps indicated, as elucidated in the fig. (1). If the scoring matrix is represented by H, then the value,

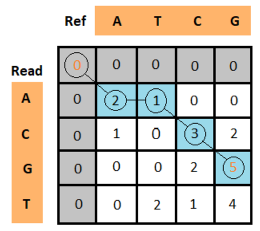

Figure 1: Scoring Matrix with $\delta(r, q) = 2$, -1 and $w = 1$. Ref. alignment: A T C G Read alignment: A _ C G

$$H_{i,j} = max[H_{i-1,j} - w,\ H_{i,j-1} - w,\ H_{i-1,j-1} + \delta(r, q),\ 0],$$

and the pointer matrix used while traceback stores the argument from which maximum is coming. In this project, a modified version of this algorithm - the Banded Smith-Waterman algorithm is used[1]. In it, instead of computing the entire scoring matrix, only the scores in a band about the diagonal are computed. This is justified based on the observation that the optimal alignment almost always lies close to the diagonal.

2 Review of existing work

There have been quite a few papers that propose hardware implementations of genome sequencing algorithms. We primarily focus on the work by Liao et al., titled "Adaptively Banded Smith-Waterman Algorithm for long reads and its hardware accelerator"[2]. This paper proposes an adaptively banded Smith-Waterman Algorithm that is hardware compatible. Based on the observation that the optimal alignment paths lie close to the diagonal, the paper justifies the use of the banded Smith-Waterman algorithm, which computes the cell scores only in a band about the diagonal, in contrast to other algorithms like GACT[3] that take into account the entire space. The paper claims that this approach doesn't result in any significant loss in accuracy while having substantial gains in terms of speedup. From the hardware perspective, most of the previous works on the banded Smith-Waterman involve performing the score computing part on the hardware, and the traceback part on the software. This is one of the few works where the traceback part is also performed on the hardware. Though the paper also proposes a dynamic overlapping heursitics in the algorithm, only Banded Smith Waterman part is chosen to build a hardware accelerator. A processing element (PE) array based architecture has been proposed by the authors, for parallel computations of the score matrix, and a dedicated traceback module for the traceback phase. We restrict our review primarily to the Banded Smith Waterman algorithm. Most of the other works on this algorithm have similar ideas of parallelising the computations across the anti-diagonal direction and achieving a better data-flow using systolic architecture, and have this work as the starting point. Some of them, like the paper by Li et al.[4], go on to implement pipelining between the scoring and traceback phase, for better throughput and re-use of PEs. Here on, we shall implement the algorithm by Liao et al., and analyse the arguments put forth by the authors.

3 Baseline Code

The Banded Smith Waterman Algorithm proposed by Liao et. al.[2]. was implemented completely on software for analysis, using Python language[5]. The code accepts long reads as inputs, and performs alignments on these using a left extend approach. The subsequence length (L) and band width (B) were chosen as 8 and 4 respectively. The code starts aligning R and Q subsequences of length L, starting from the right, and proceeds with independently aligning the next set of subsequences towards the left. This part is realised using the BandedSW() function. Before the aligned subsequences are appended, they are checked for dynamic overlap - performed by the dynamic_overlap() function. The Python code was run on Google Colab with a few test sequences. The correct functionality of the code was determined by comparing the results for the test sequences generated by our code with the results given by an online tool[6], which computes the same using high performance BLAST algorithm. Once the correct functionality was determined, the code was instrumented to profile the time taken by different segments of the code. The results obtained, averaged over 10 different test sequences are summarised in fig. 2(a). Clearly, the most significant computational time in the algorithm is that of the BandedSW function - that corresponds to the scoring and traceback phases. Further, we see that there is a scope for parallelism in it. Therefore, it makes sense to accelerate this part, as suggested in the paper. Further, analysis of the sub parts of BandedSW function shows that the scoring takes the major chunk, while the traceback is a very minor part. This shall be looked into later.
Due to resource constraints, the designed accelerator could only be integrated with the picoRV32 processor, rather than a Intel PC as mentioned in the paper. In order to ensure that we have meaningful comparisons, we deployed the software code for the BandedSW function on the picoRV32 processor and analyzed the same.

Complete code	Time (ms)	BandedSW()	Time (ms)
BandedSW	4.249	Scoring	0.658
Dynamic Overlapping	0.053	Traceback	0.026
Total time	5.178	-	-

PicoRV32	Cycles
B, L = (04, 08)	7132
B, L = (08, 16)	28287
B, L = (16, 32)	105374

Figure 2: Profiling the Baseline Code - (a) Python Code profiled on Google Colab (b) BandedSW function profiled on picoRV32 processor

3.1 Analysis of the software implementation on picoRV32

For the same case of $L = 8$ and $B = 4$, we could clearly establish that using the banded algorithm rather than the complete algorithm reduced the cycle count for the scoring phase from 7750 cycles to 6201 cycles on the picoRV32 core - which is due to the reduction in computation from 64 cells to 47 cells. With the BandedSW algorithm adopted, the traceback phase takes 931 cycles while the scoring takes 6201 cycles, and this holds true given the complexity of traceback which goes as length of alignment, while the scoring is second-order in terms of the length of sub-sequences. Further, note that for computing a given cell, say, $(1, 1)$ in fig. 1, $(0, 0)$ is required. So we need to store the value of score for about B computations before it is used by the last required cell. This requires maintaining a score matrix of size proportional to the square of length of sub-sequences. The cycle counts for different combinations of input sequence lengths have been presented in the Fig. 2(b), where it can be verified that as the B and L are doubled, number of cycles increase by around 4 times, indicating a non-linear complexity. We thus see two benefits of moving this function to hardware : (1) the anti-diagonal elements can be processed parallely on processing elements, and (2) by aligning the PE front along the anti-diagonal line in a systolic array architecture, the inter-computation dependencies can be easily satisfied using intermediate registers. Although in comparison to scoring phase the traceback module is not compute heavy, implementing the same as well on hardware has advantages in terms of memory transfers, which will be analysed at a later point. Even though the baseline code could have potentially been software pipelined, it isn't quite easy to implement for this case, and analysing the same on picoRV32 wouldn't be quite possible.

4 Hardware Implementation

The systolic array architecture proposed in Liao et. al.[2]. was implemented to realise the scoring and traceback phases on hardware. The independent computations along the anti-diagonal line are implemented parallely with B PEs (maximimum 4 (B) independent computations in parallel). The architecture for the case of 4 PEs has been presented in fig 3(a). The scoring phase is broadly divided into 3 regions - the top and the bottom-corner region, where only some PEs are active, and the middle band extension region, where all the PEs are active as shown in fig 3(b). In the top and bottom corner region of operations, the PE front shifts rightward continuously (on the matrix), while in the middle region, we have alternate rightward and downward shifts. A rightward (downward) shift is implemented in the architecture as shifting only the $R(Q)$ shift-register (which feed in inputs to the PEs) - the $Q(R)$ inputs to the PEs remain unchanged.

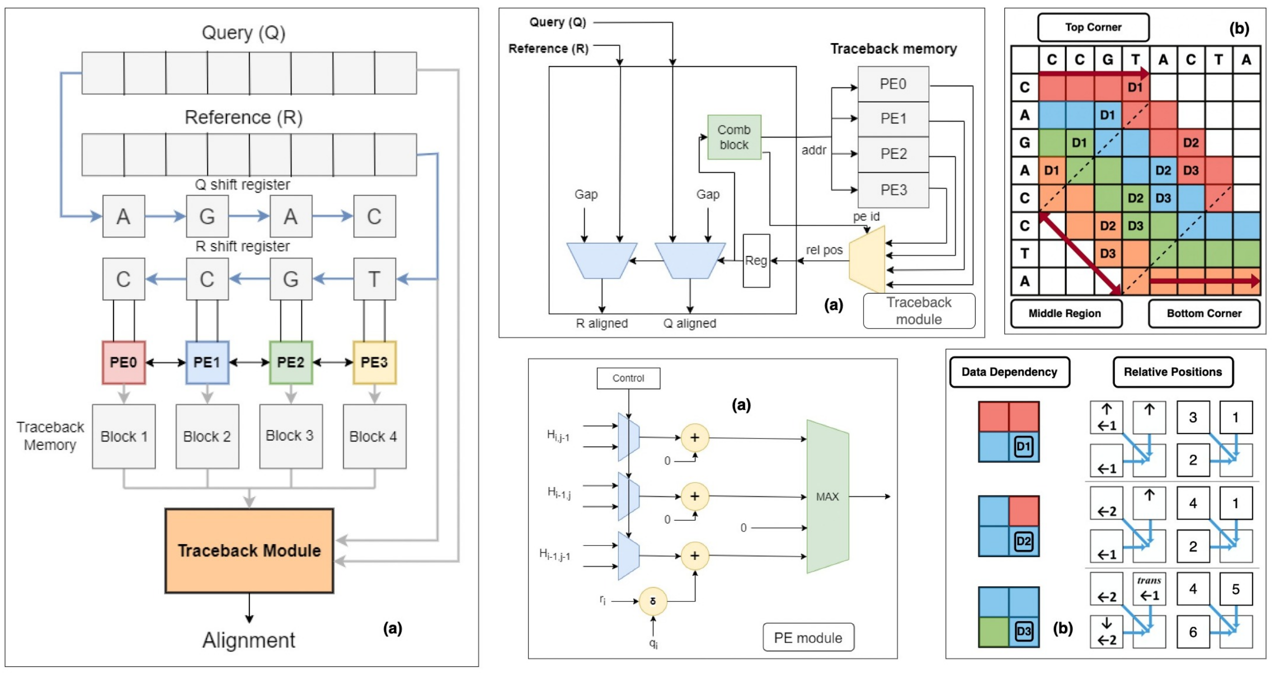

Figure 3: (a) Accelerator architecture - the figure on the left shows the overall architecture, with Q and R shift registers, 4 processing elements, corresponding traceback memory and the traceback module. The traceback module and processing element have been further elaborated on the right (b) There could be 3 types of data dependencies (D1, D2, D3) that a PE could face. This determines how the traceback should point to a cell in the traceback memory. For instance, consider the blue D1 cell in the figure. Suppose the traceback direction here is an upward diagonal - pointing to the cell (0,1). This implies that cell was computed by PE$_1$, in the previous cycle. Hence, to access that location, we have to access the data stored in (PE$_{id-1}$) at the (last_stored_address - 1) corresponding to it. Similarly, the other such dependencies could be figured out. There could be 6 combinations which needs a 3-bit encoding of the traceback pointer, as indicated in the relative position part.

The above architecture takes B cycles to initialise the Q register. The total computation involves $(B - 1)$ rightward shifts initially, followed by $2(L - B)$ alternate downward and rightward shifts, then finally followed by $(B - 1)$ rightward shifts. The PE essentially computes the entries of the traceback matrix - a 3-bit relative position corresponding to the maximum score, and stores it in an intermediate traceback memory. Since the traceback memories have been defined to be separate for each PE, the entries should clearly be able to point to the PE (and the address in its memory) in which the next aligned entry would be. This encoding scheme in effect symbolises the data dependency scheme in the 3 regions of operation, as described in fig. 3(b). Note that now because of the parallel computations, a PE would require the results of other PEs that were computed not more than a cycle back. Thus, we only require to maintain the outputs of PEs for 2 cycles - and this can easily be done using a pair of shift registers.

Once the trace-back memory has been filled by the PEs, the trace-back module is invoked using the start_traceback signal. Starting from the bottom right cell, it reads the data from the traceback memory, and performs the trace-back by successively shifting to locations pointed to by the current relative position, till it hits the null pointer. At each cycle, the traceback module gives out an aligned base pair each for R and Q, and these are loaded to a register array. The architecture of the trace-back module is presented in fig 3(a). Considering a buffer of 2 cycles between traceback and scoring, the traceback takes L_{al} cycles, which is the length of alignment. Summing up the previous cycle count for scoring, given the inputs the hardware takes $C = B + 2L + L_{al}$ cycles to compute the output.

Finally, when the traceback module completes its operations, the ready signal is passed by the accelerator, signalling the core to read the results. This entire architecture was implemented in Verilog HDL for the case of $L = 8$ and $B = 4$, and the same was simulated, tested for correctness and integrated with picoRV32. Post synthesis simulation was also performed, and the entire accelerator was implementable on a Kintex 7-T platform. The hardware resource utilization summary and timing reports can be found in the repository.

4.1 Analysing the Implementation integrated with picoRV32

From the accelerator design, we know that given the inputs to the accelerator as well as the start signal, it takes 30 cycles for the result to be ready (taking L_{al} as 10 including null pointer). By instrumenting the code, we find that sending the data to the inputs of accelerator from the picoRV32 core takes 26 cycles on software side. Post which - setting the reset signal, processing by the accelerator, and reading the ready signal by the core after completion takes 97 cycles (involves loop overhead and comparisons also). Finally, getting back the results from the core takes 22 cycles. So the whole computation takes 145 cycles, while on the other hand, the baseline software implementation took 7132 cycles. If the initial core and accelerator integrated core are assumed to be run at the same clock, this indicates a speed of about 50x. Upon performing synthesis on a Kintex- 7T FPGA, it was found that the core alone could be run at a maximum clock of $\sim$400 MHz, while the integrated design could be run at $\sim$166.67 MHz. Thus, this effectively implies a speed-up of around 20x.

As the traceback phase isn't quite compute intensive, and takes approximately only $\frac{1}{10}th$ of the time taken for scoring, it may appear that it makes more sense to perform this on software. Infact, some of the initial works in this field have been performing traceback on software. However, upon a closer look, we realise that in order to perform traceback on software, the entire traceback memory has to be read back into the core. For the realised implementation on picoRV32 - we would have to transfer $\sim$ 48 bytes (using bytes to fill traceback memory in implementation) to the core. For each serial successive read into picoRV32, it takes $\sim$ 15 cycles per word. Assuming a 1 word wide bus, this takes close to 180 cycles. On the contrary, writing back the final aligned sequences takes about $2 \times 15 = 30$ cycles. The traceback phase on hardware takes $\sim$10 cycles, and hence we gain close to 140 cycles while performing the traceback phase on hardware even without considering the cycle count on software for traceback . In doing so, we ensured that there wasn't any significant compromise on the timings - the implementation comfortably met the timing constraints of 100MHz. Further, the critical path of the traceback module alone is 2.22 ns, that of the PE module alone is 5.06 ns, while that of the integrated accelerator is 5.48 ns, and it might have been due to additional delays added to PE module, while the traceback module is not affecting the critical path.

Another aspect that justifies the hardware implementation of the scoring and traceback phase is how the numbers scale with increasing subsequence lengths. For the given architecture, the cycle count for the entire computation is of $\mathcal{O}$(L). While we implemented the design for $L = 8$, in practise higher subsequence lengths are used. On increasing L, the cycle count and memory requirement on the FPGA scales linearly. Note that increasing L would also require us to increase B by the same ratio. So on the other hand, if the alignment were performed on software, the computations would take $\mathcal{O}(L.B) \sim \mathcal{O}(L^2)$ cycles, and the memory requirement would also scale in a similar fashion. Though there appears to be a clear advantage, the flipside is that increase in B would require more hardware resources to realise the PEs. However, there are no hardware resource constraints on the FPGA, the savings on such scaled designs would be manifold.

5 Results & Conclusions

- As mentioned by Liao et al., the Python simulations performed justified that the BandedSW function in the algorithm was indeed the bottleneck.
- Using the banded algorithm instead of regular Smith-Waterman in baseline reduced the cycle count from 7750 cycles to 6201 cycles on the picoRV32 implementation, as the number of compute cells decreased from 64 to 47, which is a speed-up of 1.25x as claimed in the source.
- Baseline code took 6201 cycles for scoring phase while 931 cycles for the traceback phase. We observe that the complexity of traceback goes as length of alignment while for scoring is second order in terms of the length of the sub-sequences, with computations involved in the inner-loop being almost the same.
- The score cells across the anti-diagonal direction can be computed parallely as they are independent. This is exploited using a systolic array architecture, which also enables better data-flow taking into account inter-computation dependencies as well as independent computations as well.
- The Hardware accelerator for scoring and traceback was synthesizable on Kintex 7-T platform, and same could be run at a maximum clock of 166.67 MHz, while the core alone could be run at 400 MHz. The total computation including data-transfers to accelerator is 145 cycles, taking into account the reduction in number of cycles and decrease in clock-rate this leads, the speed-up is around 20x, and this function being run repeatedly, provides significant gains when looking at the entire sequence.
- Performing the traceback phase on hardware is justified when analysed from the data transfer perspective, as performing the traceback on software would require writing back the entire traceback memory rather than just the aligned sequences. For the implemented case, performing the traceback on hardware results in a savings of much more than 140 cycles over software, without compromising on the timing constraints.

6 Work Contribution

- Sai Gautham Ravipati, EE19B053
  Baseline Python [dynamic overlap], Baseline C, Profiling [baseline], Accelerator [pe_top.v, pe_unit.v, shift_reg.v], Debugging [functional], Documentation [code, repository], Synthesis [yosys], Report [Sections - 3.1, 4, 4.1 (partial), Figures - 2, 3b].
- Shashank Nag, EE19B118
  Baseline Python [bandedsw], Accelerator [traceback.v, bsw_acc.v (minor)], Integrating picoRV32 [axi4_periph.v], Debugging [functional], Profiling [Integrated accelerator], Documentation [code, repository], Report [Sections - 3, 4.1, 5 (partial)].
- Vishnu Varma V, EE19B059
  Baseline Python [main], Accelerator [bsw_acc.v], Debugging [syntactical], Integrating picoRV32 [hello.c], Documentation [repository], Synthesis of the design and timing report [Vivado], Report [Sections - 1,2,5, Figures - 1, 3a].

7 Code & Demo Video

The repository with the code for this project could be found at `https://github.com/sgauthamr2001/FPGA_Genome_Alignment`, and the demo video for the project could be found at `https://bit.ly/ee5332project`

8 References

The following items have had a major contribution in shaping up this project. In addition to these, several other papers and articles were reviewed while going about this project.

1 : Aligning two sequences within a specified diagonal band, Chao et al.

2 : Adaptively Banded Smith Waterman algorithm for long reads and its hardware accelerator, Liao et al.

3 : Darwin: A hardware-acceleration framework for genomic sequence alignment, Turakhia et al.

4 : PipeBSW: A Two-Stage Pipeline Structure for Banded Smith-Waterman Algorithm on FPGA, Li et al.

5 : A part of the baseline Python code for BandedSW was adapted from here

6 : `https://blast.ncbi.nlm.nih.gov/Blast.cgi`