

DEVELOPING A HIGHLY PARALLELIZED TCR  
SYNTHESIS ALGORITHM ON GPGPU AND FPGA FOR  
ACCELERATING THE STUDY OF IMMUNE SYSTEMS

by

Elnaz Tavakoli Yazdi

---

Copyright © Elnaz Tavakoli Yazdi 2018

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

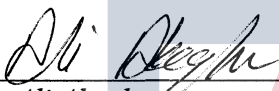
In the Graduate College

THE UNIVERSITY OF ARIZONA

2018

THE UNIVERSITY OF ARIZONA  
GRADUATE COLLEGE


As members of the Master's Committee, we certify that we have read the thesis prepared by **Elnaz Tavakoli Yazdi**, titled ***Developing a highly parallelized TCR synthesis algorithm on GPGPU and FPGA for accelerating the study of immune systems*** and recommend that it be accepted as fulfilling the thesis requirement for the Master's Degree.

  
\_\_\_\_\_  
Ali Akoglu

Date: 11/14/2018

  
\_\_\_\_\_  
Tesiron Adegbiya

Date: 11/14/2018

  
\_\_\_\_\_  
Salim Hariri

Date: 12/12/2018

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to the Graduate College.

I hereby certify that I have read this thesis prepared under my direction and recommend that it be accepted as fulfilling the Master's requirement.

  
\_\_\_\_\_  
Ali Akoglu,  
Associate Professor,  
Electrical and Computer Engineering Department  
University of Arizona

Date: 12/10/2018

## ACKNOWLEDGEMENTS

I would like to thank all the people who stood by my side during my masters studies. First of all, I would like to express my gratitude to my advisor. I am very thankful to Prof. Ali Akoglu for his advice, guidance, comments, and support during this path. Without him, this work would never have been accomplished. I have learnt a lot from him during past two years and his teachings will be always appreciated throughout my career.

I would like to thank Dr. Adam Buntzman who helped me with understanding the medical aspect of the DNA recombination process.

I would like to thank my beloved husband (Mohsen Bahrami) who has been very patient, supportive, helpful, and understanding throughout this path. Without his supports, it would be almost impossible to start this path. Last but not least, I would like to express my gratitude to my parents who has been very supportive during past two years.

## DEDICATION

*I dedicated this work to my family...*

## TABLE OF CONTENTS

LIST OF FIGURES . . . . .	7
LIST OF TABLES . . . . .	8
ABSTRACT . . . . .	9
CHAPTER 1 INTRODUCTION . . . . .	11
CHAPTER 2 The DNA Recombination Algorithm . . . . .	16
2.0.1 Biological Perspective . . . . .	16
2.0.2 Algorithmic Perspective . . . . .	18
2.0.3 Input data set . . . . .	19
CHAPTER 3 The GPU-based Implementation of DNA Recombination Algorithm . . . . .	21
3.1 The Parallelization Strategy . . . . .	21
3.2 The Bit-Wise Implementation . . . . .	23
3.2.1 Conversion of input data set . . . . .	24
3.2.2 GPU Kernel . . . . .	25
3.3 The Multi-GPU Implementation . . . . .	29
3.4 Experimental Setup . . . . .	30
3.5 Experimental Results . . . . .	31
3.5.1 Thread Block Configuration Analysis . . . . .	31
3.5.2 Bit-wise Simulation Results . . . . .	34
3.5.3 Multi-GPU Simulation Results . . . . .	34
CHAPTER 4 FPGA-based Implementation of The DNA Recombination Algorithm . . . . .	40
4.1 Hardware Implementation of N Level Parallelization . . . . .	40
4.1.1 N Level Parallelization Strategy . . . . .	41
4.1.2 Processing Unit . . . . .	43
4.1.3 n-sequence Initiator Unit . . . . .	46
4.1.4 In vivo Address Generator Unit . . . . .	47
4.1.5 Processing Element . . . . .	48
4.1.6 Address Generator Unit . . . . .	49
4.2 Simulation Results of the N Level Parallelization Architecture . . . . .	49
4.3 Drawbacks of N Level Parallelization Architecture . . . . .	50
4.4 Hardware Implementation of VJ Level Parallelization . . . . .	52
4.4.1 VJ Level Parallelization Strategy . . . . .	52

TABLE OF CONTENTS – *Continued*

4.4.2	Combination Unit . . . . .	56
4.4.3	Comparison Unit . . . . .	58
4.4.4	First In First Out Buffer (FIFO) . . . . .	62
4.5	Simulation Results of the VJ Level Parallelization Architecture	64
4.6	Drawback of VJ-Level Parallelization architecture . . . . .	64
4.7	Hashing Functions . . . . .	67
4.7.1	The class of function $H_3$ . . . . .	68
4.8	Hardware Implementation of VJ-Level Parallelization Using Hash Function . . . . .	68
4.8.1	Combination Unit . . . . .	69
4.8.2	Hash Unit . . . . .	69
4.8.3	Comparison Unit . . . . .	69
4.9	Simulation Results of the VJ Level Parallelization Architecture Using Hash Function . . . . .	70
CHAPTER 5	Conclusion and Future Work . . . . .	74
5.1	Conclusions . . . . .	74
5.2	Future Work . . . . .	75
References	. . . . .	77

## LIST OF FIGURES

2.1	Brief view of the $V(D)J$ recombination process. The figure shows p-nucleotide formation with length one for $V-$ gene termini, two for left side of $D-$ gene termini, four for right side of $D-$ gene termini and four for $J-$ gene termini in step one. Example depicts elimination of one nucleotide on the $V-$ gene termini, three nucleotides on both side of the $D-$ gene and two nucleotides on $J-$ gene termini. . . . .	17
3.1	A Normalized distribution of <i>in vivo</i> sequences across 240 VJ pairs. . . . .	23
3.2	Brief view of the comparison process for the bit-wise version of $V(D)J$ recombination process. . . . .	28
3.3	Normalized results of four different thread per block configurations (32, 64, 128, 256) for n-nucleotide length of four to nine. . . . .	32
4.1	The hardware implementation of $V(D)J$ recombination process using the N-level parallelization approach. . . . .	42
4.2	The resource utilization of the proposed architecture for different values of the $D_P$ . . . . .	45
4.3	The critical path delay of the proposed architecture for different values of the $D_P$ . . . . .	45
4.4	A high level view of hardware implementation of VJ level parallelization for the $V(D)J$ recombination process. . . . .	55
4.5	The algorithmic description for the structure of Combination unit. . . . .	59
4.6	The algorithmic description for the structure of Comparison unit. . . . .	63
4.7	A high level view of hardware implementation of VJ level parallelization with hash unit for the $V(D)J$ recombination process. . . . .	71

## LIST OF TABLES

3.1	The total number of unique n-nucleotide sequences based on the length of n-nucleotide . . . . .	24
3.2	2-bit encoding scheme . . . . .	25
3.3	P100 GPU Streaming Multiprocessor Resources . . . . .	30
3.4	GPU resource utilization for n-nucleotide length of seven with four different thread block configurations. . . . .	33
3.5	The memory footprint for the bit-wise implementation in comparison with the baseline approach. . . . .	35
3.6	Execution time on single GPU: Baseline vs. Bit-wise Implementations . . . . .	35
3.7	Execution Time for the bit-wise implementation on different number of GPUs. . . . .	36
4.1	The total number of unique n-nucleotide sequences and total number of possible combinations of $n_1Dn_2$ with a given $D$ sequence based on the length of n-nucleotide. . . . .	44
4.2	Resource Utilization for a single CU . . . . .	44
4.3	FPGA Resource Utilization for the N Level Parallelization Architecture. . . . .	51
4.4	Execution time for the FPGA and bit-wise GPU-based implementations using Virtex-7 and P100 GPU, respectively. . . . .	52
4.5	Resource Utilization for a single PU . . . . .	53
4.6	The total number of <i>in silico</i> sequences with the same features for a given n-nucleotide's length with and without the $D$ gene. . . . .	61
4.7	Execution time for the n-level and VJ level parallelization approach for FPGA-based implementation using Virtex-7 in comparison with the bit-wise GPU-based implementations using P100. . . . .	65
4.8	FPGA Resource Utilization for the VJ Level Parallelization Architecture. . . . .	66
4.9	Execution time for the VJ level parallelization approach with and without hash function for FPGA-based implementation using Virtex-7 in comparison with the bit-wise GPU-based implementations using P100. . . . .	72
4.10	FPGA Resource Utilization for the VJ Level Parallelization Architecture Using Hash Function. . . . .	73



# ABSTRACT

The  $V(D)J$  recombination process is the primary mechanism for generating a diverse repertoire of T-cell receptors (TCRs) essential to the adaptive immune system for recognizing a wide variety of diseases. The diverse set of TCR are required by the immune system to increase the chance of accurate identification of the foreign invaders, which results in successful recovery from diseases. Furthermore, analysis of the TCR repertoire helps immunologists to understand the functionality of immune system in presence of different diseases and find the correlation between diseases and immune receptors. However, modeling this diverse TCR repertoire is computationally challenging as the total number of TCRs to be generated and processed can exceed  $10^{18}$  sequences. This massive scale of data processing poses as the barrier for immunologists to successfully understand the functionality of human immune system. Therefore, reducing the timescale of modeling the TCR repertoire will help immunologists to test their assumptions and solve their fundamental questions.

In this study, we propose FPGA and GPU-based implementation of  $V(D)J$  recombination process for accelerating the analysis of TCR repertoire. For the GPU-based implementation, we propose a *bit-wise* implementation of the  $V(D)J$  recombination algorithm, which reduces the memory footprint and the execution time by factors of 4 and 2 respectively compared to the current state-of-the-art GPU-based implementation. We devise an encoding procedure to convert the input data set from character based domain to binary domain and pack a sequence of four characters into a single byte for the bit-wise implementation. We also proposed new *indexing* scheme for addressing input data that are not aligned with the byte-addressing. We present a multi-GPU implementation, experimentally identify suitable workload partitioning strategies for both single- and multi-GPU implementations, and finally expose the relationship between workload size and limited scalability offered by the

algorithm on a cluster with up to eight GPUs. We show that the bit-wise implementation reduces the execution time from a time scale of 40.5 hours to 18.9 hours on a single GPU and to 4.3 hours on a 8-GPU configuration.

For the FPGA-based implementation, we first utilize the N-level parallelization approach that is used for the GPU-based implementation. Simulation results show that this approach does not perform as expected for the FPGA-based implementation of  $V(D)J$  recombination process due to the communication overhead between FPGA components. Therefore, we propose the VJ level parallelization approach to reduce the communication among components. We show that the VJ-level architecture reduces the execution time by a factor of 2.34 in comparison with the N-level parallelization approach.

## CHAPTER 1

### INTRODUCTION

Adaptive immune system protects vertebrates by detecting and neutralizing foreign invaders (antigens) using T-cell receptors (TCRs), which are placed on the surface of a T-cell [1]. A T cell is a sub-type of white blood cell that plays a central role in recognizing and neutralizing fragments of antigen with the TCRs. A TCR recognizes an antigen by detecting the small protein fragments that are on the surface of that antigen, and then sends a message to the nucleus of its T-cell. This successful recognition induces a response for eliminating antigens [2]. The diversity in the TCR pool increases the chance of detecting a variety of antigens for the adaptive immune system, which is the first step of a successful recovery from diseases. Analysis of TCR pool (repertoire) is crucial for understanding the functionality of healthy immune system, determining the nature of successful and unsuccessful immune responses, and understanding the immune mechanism in presence of different diseases such as type I diabetes, various cancers (blood, breast, colorectal, etc.), rheumatoid arthritis (autoimmune disease), and multiple sclerosis [3]. The response of immune system to specific antigen often leaves evidence in the form of repertoire sequence patterns (signatures) that are common across individuals and these signature patterns can be associated with the corresponding antigen. Identification of these signatures help biologists to understand the correlation between the immune receptors and different disease, which provides researchers with the ability to identify immune receptor clones that can be converted into precision vaccines [4–6].

A diverse set of TCRs is required for the adaptive immune system to successfully detect wide variety of antigens. This diversity is achieved by the immune systems of the vertebrates through the DNA recombination process, which is known as the  $V(D)J$  recombination [7]. This process involves rearrangement of variable ( $V$ ), diversity ( $D$ ), and joining ( $J$ ) gene segments in a

combinatorial way chosen from members of each gene family [7, 8]. The form and length of each gene segment varies across different species, and it is more complex in the human than vertebrates. For example, there are 20 different  $V$  genes in the mice, while there are 50 different  $V$  genes in human [9].

In the clinical environment, ability to predict the repercussion of immune system to different antigens is a challenging research problem [9–13]. For such prediction capability, immunome researchers need a way to model all possible unique TCR sequences (TCR repertoire) that can be generated through the recombination process. This would allow the researchers count the number of times each unique sequence is created through different recombination paths, which forms the baseline for statistical analysis on correlating a specific TCR chain with a specific antigen. However, modeling of all  $\alpha\beta$ TCR sequences is considerably complex, since the potential TCR repertoire of mouse contains more than  $10^{15}$  sequences [14]. Furthermore, the total number of paths exhausted to generate all the sequences is expected to exceed  $10^{18}$ . This massive scale of data processing poses as the barrier for immunologists, which has led them to computationally tractable statistical methods [15], [16] with a trade off in accuracy [15], [16]. Reducing the timescale of modeling all TCR $\beta$  chains will enable immunologists to test the validity of their assumptions and solve their fundamental questions rapidly such as: What is the real size of the TCR $\beta$  repertoire? Is the procedure of generating TCRs random or biased? What can we say about the frequency distribution of various TCR sequences? The most exhaustive study on TCR synthesis to this date [17] models more than  $10^{14}$  TCR $\beta$  chains by exploiting the data parallel nature of the recombination process and mapping the algorithm entirely on a single graphic processor unit (GPU). This study shows that the execution time can be scaled down to 13 days using an NVIDIA GTX 480 from 52 weeks on a single general purpose processor. This is the first study that models all the potential TCR $\beta$  repertoire of the mouse in which the number of recombination pathways exceeds  $4 \times 10^{14}$ .

For the human system, diversity of the TCR repertoire shows variations from one person to another and increases significantly such that there are

more than 50 million TCRs in the immune system of 100 humans. From mouse data set to human data set, the scale of the data increases from  $10^5$  to  $10^7$ . Furthermore, in humans there are 50 basic  $V$  genes, 2 basic  $D$  genes, and 13 basic  $J$  genes, while in the mouse data set there are 20  $V$  genes, 2 basic  $D$  genes, and 12  $J$  genes. From mouse to human, the TCR repertoire increases by three orders of magnitude to  $10^{18}$  [17]. To be able to cope with the scale of the simulations at  $10^{18}$  level, our aim in this study is to investigate ways to reduce the execution time and memory footprint of the recombination process so that we can rapidly model systems that are more complex than the mouse. For this, we make the following contributions:

- Devising a bit-wise GPU-based implementation of the recombination process, which consists of fine-grained shift, concatenation, comparison, and counting over the binary domain input data set.
- Extending the bit-wise implementation to a multi-GPU environment and evenly distributing workload across the GPUs being used.
- Utilizing the N-level parallelization approach that is used for the GPU-based implementation and implementing the  $V(D)J$  recombination algorithm on the field programmable gate array (FPGA).
- Devising a VJ-level parallelization method for the FPGA-based implementation to overcome the draw back of N-level parallelization method.

In order to utilize the bit-wise operations, we devise an encoding procedure to convert the input data set from character based domain to binary domain (2-bits per character) and pack a sequence of 4 characters into a single byte. The bit-wise representation of input data set reduces memory access time by a factor of 4, since we can fetch four characters with one memory access. After mapping the input data set to binary domain, we pad the end of input data sequences whose length are not divisible by eight with 0's. Correspondingly, we develop a new indexing scheme for addressing the input data stored in the constant memory of GPU to avoid using padded bits of the input data that are not aligned with the byte addressing. Then, we introduce a *task*

*generation* function that generates a unique task for each thread based on its identification number, ensures a unique path for a given sequence by each thread and eliminates the communication between the GPU threads. The two-bit based representation reduces the global and constant memory footprint by factors of 4 and 3.5 compared to the baseline GPU implementation [17]. Utilizing bit-wise operations and reducing the amount of data transfers reduces the execution time by a factor of 2 using an NVIDIA Tesla P100 GPU. CUDA natively supports 32-bit integer shift and 32-bit bitwise "AND", "OR", and "XOR" operations [18], which allows us to implement the bit-wise version of recombination process as it heavily relies on shift, concatenation( using "shift" and "or" ) and comparison ("xor") operations.

For the multi-GPU implementation, we define an *indexing* function, which generates global index for threads in different GPUs based on the index of GPU, and the GPU dimension. Then, we use the proposed *task generation* function to generate unique tasks for each GPU thread. Overall, our aim is to count number of times each *in vivo* sequence can be generated artificially by rearranging input data set ( $V$ ,  $D$ ,  $J$  genes and  $n$ -nucleotide sequence) to model the TCR repertoire. Therefore, each thread works on its assigned task based on its global index, repeatedly generates a unique sequence, increments the counter for that sequence if it exists in the *in vivo* data set till that thread completes the task of generating all possible sequence based on its given  $n$ -nucleotide. The multi-GPU implementation reduces the execution time from a scale of 18.9 hours to 4.3 hours using 8 GPUs.

As stated above, the recombination process involves intensive amount of fine-grained shift, concatenation, comparison, and counting operations over a data-set generated on the fly. We take advantage of the fine grained parallelism offered by the FPGA whose architecture naturally matches the program architecture of the recombination process. Therefore, we first map the  $V(D)J$  recombination algorithm onto the target FPGA using the GPU-based parallelization approach, which is known as N-level method. In order to address this, we first define the degree of parallelism for the final hardware architecture. We also show the relation between degree of parallelism with critical path delay

and resource utilization. We form the memory hierarchy based on the selected degree of parallelism and design required components to map the recombination process. In addition, we utilize the length and  $VJ$  pair index of the generated sequence to pare down the comparison search space. Experimental results for the FPGA implementation show that the N-level parallelization approach causes communication overhead among FPGA components and result in poor performance. Therefore, we devise a VJ-level parallelization approach to form a unique memory hierarchy to match the data access patterns for various stages of the algorithm, which results in the elimination of communication overhead among components. We show that the VJ-level parallelization approach accelerate the recombination process by a factor of 2.34 in comparison with the N-level method for the FPGA-based implementation.

The rest of this dissertation is organized as follows: Chapter 2 provides an overview of the DNA recombination algorithm from biological and algorithmic perspectives. Chapter 3 describes a bit-wise implementation of the recombination process on a single and multi-GPU. A detailed explanation of the FPGA-based implementation of the  $V(D)J$  recombination process is provided in Chapter 4. Conclusions and future work are discussed in Chapter 5.

## CHAPTER 2

### The DNA Recombination Algorithm

The  $V(D)J$  recombination process is a specialized DNA rearrangement critical to the adaptive immune system. In this section, we describe the DNA recombination process from biological and algorithmic perspectives, and highlight key features related to our parallelization approach. We also provide detailed explanation about the structure of input data set used in our experiments.

#### 2.0.1 Biological Perspective

The TCRs are created by recombination of the  $V$ ,  $D$ , and  $J$  gene segments. Fig. 2.1 illustrates the recombination process using an example sequence formed by the  $V$ ,  $D$  and  $J$  segments. The two rows in step 0 represent the two complementary DNA strands: the *template strand* and its mirror image the *coding strand*. As the  $V$ ,  $D$ , and  $J$  segments go through the recombination process for generating unique sequences in search of a sequence that matches the antigen, diverse set of sequences are generated. There are three critical steps that contribute to this diversity, which we summarize by highlighting the core factors in the following paragraphs.

In the first step, the recombination activation gene, *recombinase*, cuts the DNA at the joints between  $V$  and  $D$  segment pairs, and  $D$  and  $J$  segment pairs. Immediately, the *template strand* and *coding strand* bind to each other where the cut occurs. Subsequently, the *Artemis exonuclease* enzyme releases circular ends irregularly to generate a palindromic nucleotide (p-nucleotide) with variable lengths [7, 19, 20]. From the starting point of the arrow shown in step one of Fig. 2.1, up to length of four genes from the coding strand are appended to the template strand on its right termini for the  $V$  segment, both termini for the  $D$  segment, and left termini for the  $J$  segment. This p-nucleotide addition of length up to four is one of the major contributors to the diversity during the recombination process.



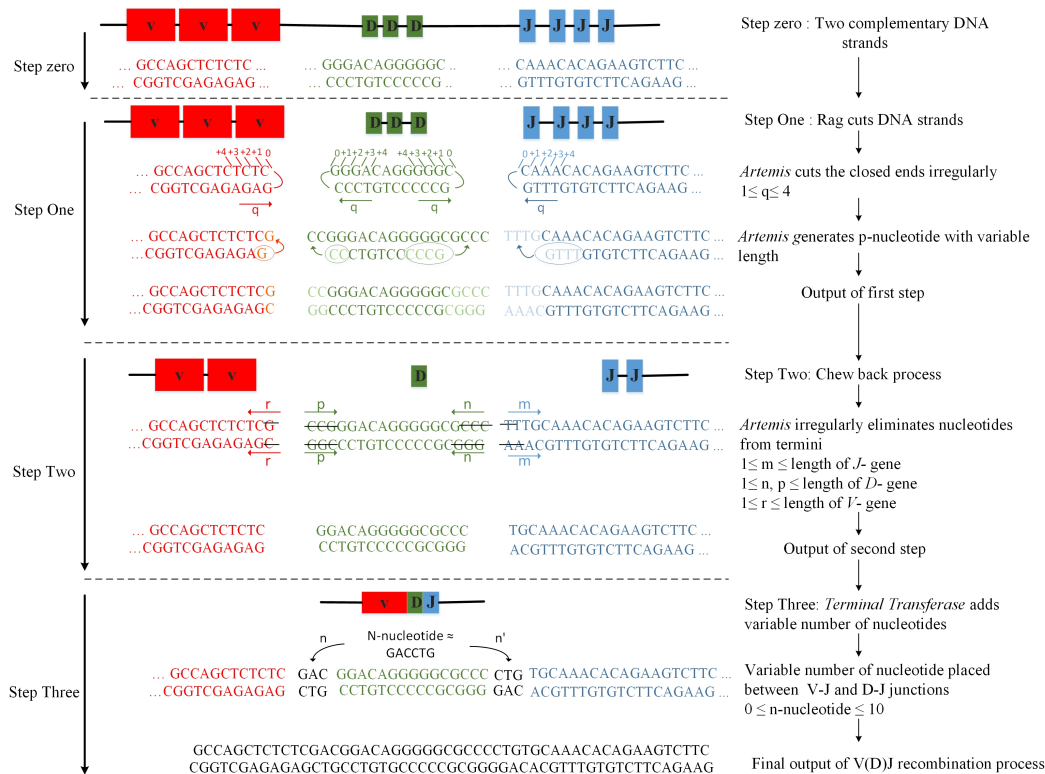


Figure 2.1: Brief view of the  $V(D)J$  recombination process. The figure shows p-nucleotide formation with length one for  $V$ - gene termini, two for left side of  $D$ - gene termini, four for right side of  $D$ - gene termini and four for  $J$ - gene termini in step one. Example depicts elimination of one nucleotide on the  $V$ - gene termini, three nucleotides on both side of the  $D$ - gene and two nucleotides on  $J$ - gene termini.

In the second step, both strands of the  $V$ ,  $D$  and  $J$  segments go through a process called chewback. Once more, the *Artemis exonuclease* enzyme is involved in this chewback process, in which a variable number of nucleotide are eliminated from the  $V$ ,  $D$  and  $J$  termini. The chewback is applied from right to left on the  $V$  segment, left to right on the  $J$  segment, and from both directions on the  $D$  segment. The amount of chewback ranges from one nucleotide to the length of that gene segment, which is the second contributor to the diversity.

In the third step, the *Terminal Transferase (TDT)* enzyme catalyzes the addition of  $n$ -nucleotides between  $V-D$  and  $D-J$  gene pairs. We consider the size  $n$  ranging between zero and ten. This range has been proven to regenerate 99.5% of the sequences in our *in vivo* data-set [17]. The *in vivo* data set has been built based on the samples that were sequenced on the Roche FLX 454 platform at the UNC-Chapel Hill High Throughput Genome Sequencing Core. The *in vivo* data set consists of 101,822 functional sequences. Finally, the DNA ligase *IV* closes off the  $V$  and  $D$  termini to form  $V-D$  junction, and  $D$  and  $J$  termini to form  $D-J$  junction. Compared to the first two factors that contribute to the diversity, having  $n$ -nucleotide addition between  $V-D$  and  $D-J$  junctions enormously grow the combinational search space, and acts as the main contributor of the diversity.

### 2.0.2 Algorithmic Perspective

We refer to the  $V(D)J$  recombination as  $VnDn'J$  recombination in this discussion about the algorithmic perspective. In this case,  $V$ ,  $D$ , and  $J$  indicate the unique sequences from each set of corresponding segments while ' $n$ ' indicates the set of all possible nucleotide combinations. We refer the generated  $VnDn'J$  sequences as the '*in silico*' sequences. Thus, to generate the *in silico* sequences, we need four inputs:  $V$ ,  $D$ ,  $J$ , and the  $n$ -nucleotide ( $n$ ) sequences.

There are four bases: A, G, C, T, used to generate a  $n$ -nucleotide sequences. Thus, for  $n$ -nucleotide length  $m$ , there are  $4^m$  unique combinations. In the recombination process, these  $n$ -nucleotide sequences can be attached on either side or on both sides of the  $D$  sequence. To differentiate between the positions,

---

**Algorithm 1:** Pseudo code for V(D)J Recombination Algorithm

---

**Input** :  $V$ ,  $J$ ,  $D$  and  $n$  – *nucleotide* sequences

**Output** : Number of times each unique *in vivo* sequence is generated (Counter)

```

1 for  $i = 0$  to number of  $V$  sequences do
2   for  $j = 0$  to number of  $J$  sequences do
3     for  $k = 0$  to number of  $D$  sequences do
4       for  $m = 0$  to number of  $n$  – nucleotide sequences do
5          $Combination =$ 
           CombineString( $V[i], n[m], D[k], n[m], J[j]$ );
6         for  $p = n - nucleotide_{length}$  to 0 do
           move ( $N[m][p] \rightarrow T[n - nucleotide_{length} - p]$ )
           for  $n = 0$  to number of in vivo sequences do
             if  $Combination == invivo[n]$  then
               Counter[n] = Counter[n] + 1;
```

---

we define the  $n$ -nucleotides as  $n$  and  $n'$ . The  $D$  sequence can cut the  $n$ -nucleotide at any position. Therefore, this complex junctional combination may lead to generation of an identical sequence through numerous ways. Our main purpose is to count the number of unique pathways that generate a given *in vivo* sequence through the recombination process. Algorithm 1 shows the pseudo code for the  $VnDn'J$  recombination process with nested loops that iterate through each  $V$ ,  $D$ ,  $J$  and  $n$  sequence to form *in silico* sequences. All single sequences are combined and stored in the variable *Combination* through the nested loops. If a generated sequence is found in the current *in vivo* set, we increment the counter value for that sequence. This process continues until the entire combinational search space is exhausted.

### 2.0.3 Input data set

The input data sets consist of  $V$ ,  $D$ ,  $J$  and *in vivo* genes. In C57BL/6 mice, there are 20 basic  $V\beta$  genes, 2  $D\beta$  genes and 12 basic  $J\beta$  genes. However, all possible patterns such as chewback and palindromic forms for each of the functional  $V$ ,  $D$  and  $J$  gene sequences need to participate in the recombina-

tion process for modeling the TCR repertoire as illustrated in Fig. 2.1. For example, the first basic  $V$  gene has a length of 14. For the  $V$  gene, up to four genes can be appended to the right end of the  $V$  gene from its mirror strand (step one, indicated as +4, +3, +2, +1), therefore the actual length of this gene can be up to 18. This would result with 18 different sequences based on the chewback process (step two).  $D$  and  $J$  gene data sets go through similar process as explained in Section 2.0.1, therefore each  $V$ ,  $D$  and  $J$  gene data set consists of several forms of sequences with different lengths. Each  $V$ ,  $D$ ,  $J$ , and *in vivo* sequence is generated using four bases (A, G, T, and C). In C57BL/6 mice, the *in vivo* data set involves 101,822 sequences, which are grouped based on the specific  $VJ$  pair used to generate that sequence. There is no other recombination path for an *in vivo* sequence other than the specific  $VJ$  pair, which generates that specific sequence. This is a key feature that we can exploit to reduce a search space within the *in vivo* data set and reduce the execution time. There are 240 such pairs since we have 20 basic  $V$  genes and 12 basic  $J$  genes in mice.

## CHAPTER 3

### The GPU-based Implementation of DNA Recombination Algorithm

We provide a detailed description regarding the GPU-based implementation of the DNA recombination algorithm in this chapter. We first evaluate two parallelization approaches for GPU-based implementation, and select the most suitable method based on their advantages and disadvantages in Section 3.1. In Section 3.2, we propose the bit-wise GPU-based implementation for the DNA recombination algorithm. We extend the bit-wise implementation to a multi-GPU environment and provide a detailed explanation in Section 3.3. Finally, we explain the experimental setup and simulation results for bit-wise implementation on a single and multi-GPU in Section 3.3 and 3.5, respectively.

#### 3.1 The Parallelization Strategy

In this section, we analyze two parallelization approaches for the GPU-based implementation of  $V(D)J$  recombination process. For each strategy, we attempt to answer the following questions: 1) What would be the workload distribution based on the parallelization strategy? 2) Does the proposed parallelization strategy result in an even workload distribution among the threads?

Since each  $V - J$  pair generates a specific sequence, the first parallelization approach would be the  $V - J$  level parallelism by assigning one  $V$  gene, one  $J$  gene, and both  $D$  genes to each thread to perform the recombination process. In this assignment, each thread needs to cover all possible  $n$ -nucleotide lengths (zero to ten) along with all possible combinations of four bases for any given  $n$ -nucleotide lengths. However, as there are 20  $V$  genes and 12  $J$  genes, this implementation would require only 240 threads and result with significantly low thread utilization on the GPU. A finer granularity of  $V - J$  level parallelism can be realized by assigning one form (refer to chewback and palindromic forms of each input gene) of  $V$  gene, one form of  $J$  gene, and both  $D$  genes to each

thread. For this approach, the total number of required threads is 102,446 since there are 362  $V$  genes, 283  $J$  genes in the chewback and palindromic forms for the mice data set. This finer level of parallelization occupies 90% of the threads on the target P100 series GPU.

In order to answer the second question for the fine-grained  $V - J$  level parallelization approach, we need to consider workload for both *combination* and *comparison* steps. We refer to the *combination* step as a process of generating all possible *in silico* sequences for a given input data and the *comparison* step as a process of comparing generated sequences with *in vivo* sequences. The workload distribution for *combination* step is even since, each thread is assigned one form of  $V$  gene, one form of  $J$  gene, and both  $D$  genes. In order to evaluate the workload distribution for the *comparison* step, we provide a normalized distribution of *in vivo* sequences across 240  $VJ$  pairs in Fig. 3.1. As shown, total number of *in vivo* sequences is not evenly distributed across different  $VJ$  pairs, which directly affects the workload of each thread. Since, each thread needs to compare its generated *in silico* sequences against every *in vivo* sequence in the corresponding  $VJ$  pair, the fine-grained  $V - J$  based assignment results in an uneven workload distribution among GPU threads.

The second solution would be n-nucleotide level parallelism, where each thread is assigned a unique n-nucleotide sequence and the recombination process is applied on that unique sequence. In this assignment, the workload for the *combination* step is even since, each thread works on one n-nucleotide sequence. In addition, the workload distribution for *comparison* step is equal among GPU threads, since GPU threads are working on the same  $V$  and  $J$  gene. Therefore, total number of *in vivo* sequence for the comparison process are equal among active threads. Since all threads share the same  $V$  and  $J$  gene pairs, this approach can take advantage of the shared memory to improve the the memory bandwidth utilization.

In order to decide which one of the stated approaches performs better in terms of execution time, we evaluate the workload per thread for the *combination* step, in both approaches. In the fine-grained  $V - J$  level parallelism, each thread generates *in silico* sequences for all possible forms of n-nucleotide

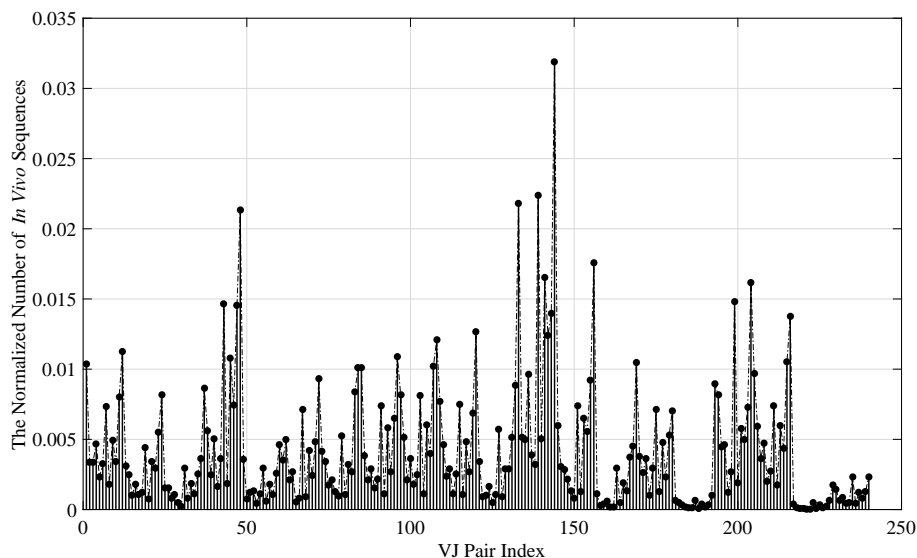


Figure 3.1: A Normalized distribution of *in vivo* sequences across 240 VJ pairs.

for length of zero to ten. Table 3.1 shows the total number of n-nucleotide sequence based on the length. Therefore, a single thread must generate 706,042,015 *in silico* sequences through the entire process as there are 1,381,717 unique n-nucleotide sequence in total and there are 505 sequences for *D* gene. In the n-nucleotide based assignment, each thread generates 51,735,230 *in silico* sequences since there are 362 *V* genes, 283 *J* genes, and 505 *D* genes. As a result, workload per thread in fine-grained *V* – *J* level parallelism is almost 13 times higher than n-nucleotide parallelization approach. In the following sections we present our approach to bit-wise and multi-GPU implementations based on n-nucleotide level parallelism.

### 3.2 The Bit-Wise Implementation

Mainly, there are two phases for the implementation of bit-wise representation. The first phase involves converting the input data set from character domain to binary domain, which we refer to as data conversion phase. The second phase consists of developing GPU kernel using bit-wise operations. We provide detailed explanation regarding the implementation of bit-wise version of the

Table 3.1: The total number of unique  $n$ -nucleotide sequences based on the length of  $n$ -nucleotide

<i><math>N\_len</math></i>	<i>Total number of unique <math>n</math>-nucleotide sequences</i>
0	1
1	4
2	16
3	64
4	256
5	1,024
6	4,096
7	16,384
8	65,536
9	262,144
10	1,048,576
Total	1,381,717

recombination process in the following subsections.

### 3.2.1 Conversion of input data set

The main objective of using bit-wise operations for mapping the recombination process is to reduce the memory footprint and execution time. We represent each base (A, C, T, and G) with two bits as shown in Table 3.2 and pack a sequence of four bases into a single byte. For those sequences ( $V$ ,  $D$ , and  $J$ ) whose length is not divisible by four, remainder bases will not fill the byte to its capacity. In this case, we zero pad the end of sequence such that the length of the binary string is divisible by eight (one byte). Let's consider a  $V$  sequence, which has the length of ten characters (20 bits). For this case, four zeros are appended to the end of  $V$  sequence. As a result, the new  $V$  sequence has 24 bits, which requires three bytes of data to store in the memory. The last byte of this  $V$  sequence has four zero padded bits, which we refer to as *padded bits*. We call the first two bytes, which only contain the original bits



Table 3.2: 2-bit encoding scheme

<i>Gene base</i>	<i>2-bit representation</i>
A	00
T	01
C	10
G	11

of the  $V$  sequence as *full byte*.

The maximum length of *in vivo* sequences is 60 characters (bytes). In the baseline implementation, *in vivo* sequences are padded with 0's so that the length of all sequences are equal to 64 bytes. This guarantees that the allocation of each sequence is equal to the number of threads in two warps, ensuring the memory is aligned to realize coalesced memory accesses. In the binary representation form with 2 bits per base (character), we also follow the same encoding procedure with padding, and represent each *in vivo* sequence with fixed size of 16 bytes.

In the baseline implementation, all possible forms of  $V$ ,  $D$ , and  $J$  sequences are stored in the constant memory to take advantage of the temporal locality it offers. However, the *in vivo* sequences are stored into the GPU's global memory as there are too many *in vivo* sequences ( $> 10^5$ ) to fit into the constant memory. For the bit-wise implementation, the *in vivo* data set is also stored in the global memory since the required constant memory for the binary representation of *in vivo* data set is larger than the available constant memory in a target GPU.

### 3.2.2 GPU Kernel

For the n-nucleotide level parallelism, the total number of threads is set to the total possible combinations for a given n-nucleotide sequence ( $4^m$ ), where  $m$  is the length of n-nucleotide sequence. In this case, all active threads can fetch the same input data ( $V$ ,  $D$ ,  $J$ , and *in vivo*) and each thread can apply the recombination process over its assigned n-nucleotide. This reduces the number

---

**Algorithm 2:** Pseudo code for the *task generator* function which generate a unique n-nucleotide sequence for each thread based on its thread and block indexes.

---

**Input** : *threadId*, *blockId*, and *blockDim*

**Output** : n-nucleotide sequence

$base[4] = \{00, 01, 10, 11\}$

$G_{index} = threadIdx.x + blockIdx.x * blockDim.x$

**for**  $i = 0$  **to**  $3$  **do**

**for**  $j = 0$  **to**  $9$  **increment by**  $2$  **do**

$temp = base[\{G_{index} + G_{index}/4^{4*i+(j-2/2)}\} \% 4] \ll (8 - j)$

n-nucleotide[i] |=  $temp$

---

of memory accesses (global and constant) for a specific gene sequence to one among all active threads.

As mentioned earlier, the *in vivo* sequences are partitioned into 240 groups based on the *V* and *J* gene used to generate these sequences. This feature was used to pare down the comparison search space in [17]. Indeed, *in silico* sequences are only compared against the corresponding portion of the *in vivo* data set instead of being compared with the entire data set. We also use this feature in our design. Therefore, our GPU kernel starts its execution by using *V* and *J* gene indexes to determine how many and which *in vivo* sequence will be used for the recombination process. Then each thread is assigned a unique n-nucleotide sequence based on the length of *n sequence*, *thread ID*, and *block ID*. We propose a function that generates a unique binary n-nucleotide sequence for each thread to guarantee that there is no duplicate n-nucleotide sequence. Algorithm 3 shows the pseudo code for the *task generator* function, which is used to generate a unique binary n-nucleotide sequence. After assigning a unique task to each GPU thread, the recombination process starts on the GPU.

There are four main loops in the GPU kernel. The first *for loop* iterates through each *in vivo* sequence. Upon entering this loop, threads within the block read a single *in vivo* sequence from the global memory into the shared

memory. Since, the *in vivo* sequence is shared among all threads within a block, we use *synchthread()* to assure that all threads wait until the memory transaction is completed.

The second *for loop* iterates through each *V* sequence in the current *V* gene set. All threads within a block read the same *V* sequence from the constant memory, while they work on a different n-nucleotide sequence. We compare the *V* sequence against the *in vivo* sequence. To accomplish this, we calculate the total number of *full bytes* and *padded bits* for a given *V* sequence. Then, we iterate through each *full byte* of the *V* sequence, and compare it with *in vivo* sequence one byte at a time. If there is a mismatch, we terminate the current comparison for all threads and read a new *in vivo* sequence from global memory. Otherwise, we continue on to comparing the last byte of *V* sequence with the pertinent byte of *in vivo* sequence. In order to accomplish this, we shift the corresponding byte of *in vivo* sequence to the right by the total number of *padded bits*. Accordingly, we shift that byte to the left by the same amount. We will refer to this process as an *alignment process*. Finally, we compare the last byte of *V* sequence with the aligned byte of *in vivo* sequence. This procedure is shown in step one of Fig. 3.2. If the *V* sequence completely matches with the *in vivo* sequence, we proceed to the next loop. Otherwise, we read a new *in vivo* sequence and repeat the process.

The third loop iterates through each *D* sequence. There is a difference between this loop (D-loop) and the previous loop (V-loop). The *D* sequence can cut the n-nucleotide sequence at any position as explained in Chapter 2. Therefore, each thread generates all possible combinations of  $nDn'$  sequence for a given *D* and n-nucleotide sequences. Then, they compare their  $nDn'$  sequence with *in vivo* sequence from the last character that was found to be identical to the *V* sequence in the previous loop. This is accomplished by shifting the *in vivo* sequence to the left by the length of *V* sequence. The comparison procedure is shown in step two of Fig. 3.2, and it is the same process as explained in the V-loop. If there is a mismatch between the *in silico* and *in vivo* sequence, then the thread terminates the current comparison, generates a new combination for  $nDn'$  sequence, and repeats the process.

Otherwise, we continue on to the next loop. It should be noted that, if a thread generates all possible forms of  $nDn'$  sequence for a given  $D$  and  $n$  sequence, then we load new  $D$  sequence and repeat the process.

The final loop iterates through each  $J$  sequence. In this loop, we first calculate the length of  $VnDn'J$  sequence and compare it with the length of *in vivo* sequence. If the length of *in silico* and *in vivo* sequences are not equal, then we terminate the current comparison and load a new  $J$  sequence. Otherwise, we compare the  $J$  sequence with the latter portion of *in vivo* sequence as shown in step three of Fig. 3.2. If a sequence generated by a thread matches with the *in vivo* sequence, then that thread increments the local counter stored in a register. A thread may generate the targeted *in vivo* sequence through multiple recombination paths. After all threads complete their  $n$ -nucleotide level workload, the counter value stored in the shared memory for that *in vivo* sequence is updated through reduction. At the end of this loop, reduction determines the total number of times an *in vivo* sequence is generated artificially. Finally, the first thread within the block updates the counter value in the global memory.

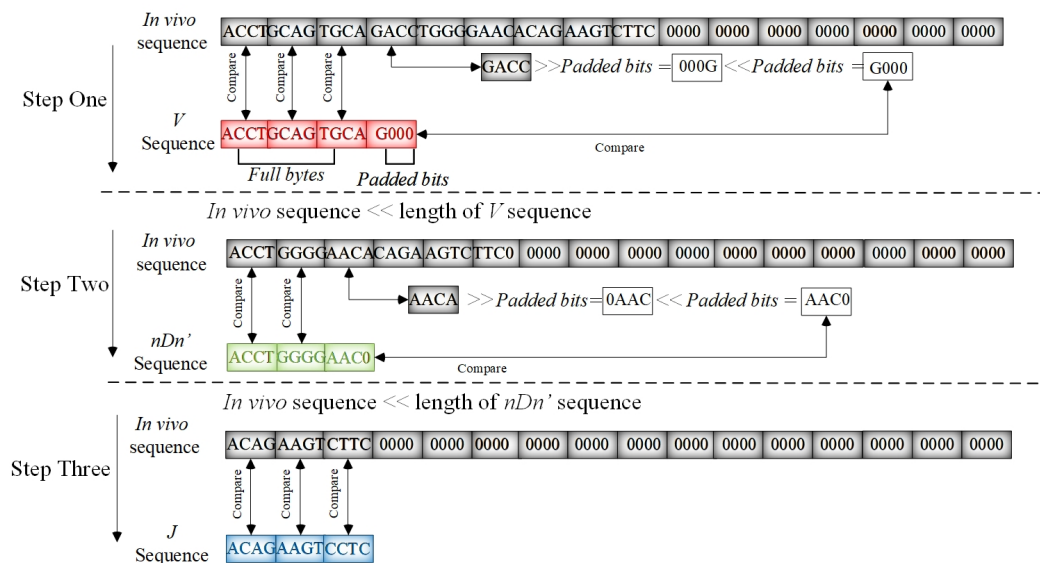


Figure 3.2: Brief view of the comparison process for the bit-wise version of  $V(D)J$  recombination process.

### 3.3 The Multi-GPU Implementation

In n-nucleotide level parallelization, threads of a GPU are assigned a unique n-nucleotide sequence while they work on the same  $V$  and  $J$  gene. From multi-GPU implementation perspective, in order to generate a unique n-nucleotide sequence for each active thread, we define a global index for each thread based on its thread Id, block Id, GPU Id, and GPU dimension as shown in (3.1) and utilize a *task generator* function that is presented in the algorithm 3.

$$G_{index} = threadIdx + blockIdx \times blockDim + GPUIdx \times GPUDim. \quad (3.1)$$

For the n-nucleotide based parallelization approach, GPU threads work on the same  $VJ$  pair so they require accessing the same *in vivo* sequences. Therefore, we replicate input data set and store it in the constant and global memories of each GPU to avoid data transfer between the GPUs.

In order to distribute the workload among GPUs, we first calculate the total number of required threads, which is  $4^m$ , where  $m$  is the length of n-nucleotide sequence. Then, we calculate the total number of required blocks based on the thread-block configuration (refer to Fig. 3.3). Finally, we calculate total number of blocks in each GPU using 3.2. For example, we need 262,144 ( $4^9$ ) threads for n-nucleotide length of 9. As we will present later in the experimental results, 128 threads per block configuration, which requires 2048 blocks in total is the desired configuration on a single GPU. Assuming that we have two GPUs, based on 3.2, each GPU is assigned 1024 blocks with 128 threads in each block. In this assignment, the workload distribution among GPUs and GPU threads are equal.

$$\#blocks = \frac{\#total\ threads - 1}{\#threads\ per\ block \times \#GPUs + 1} \quad (3.2)$$

In order to obtain the final result from multiple GPUs, we perform a reduction process [21]. This process accumulates all the results in the root node and copy them to global memory of the host.

Table 3.3: P100 GPU Streaming Multiprocessor Resources

Parameter	Value
Compute Capability	6.0
Streaming Multiprocessors (SM)	56
Threads per Warp	32
Maximum Thread Block Size	1024
Maximum Thread Blocks per SM	32
Maximum Warps per SM	64
Maximum Threads per SM	2048
Maximum 32-bit Registers per SM	65536
Maximum Registers per Block	65536
Maximum Registers per Thread	255
Maximum Shared Memory Size per SM	64 KB
Constant Memory Size	64 KB

### 3.4 Experimental Setup

We conducted our experiments on a cluster consisting of NVIDIA P100 GPU accelerators. The system is composed of 400 nodes (Intel Haswell V3 28 core processor, 192 GB RAM per node) in which 46 of them are configured as accelerator nodes with a single Nvidia P100 GPU in each node. The cluster uses FDR Infiniband for node to node interconnect and 10 Gb Ethernet for node to storage interconnect. Table 3.3 summarizes the GPU parameters. The P100 GPU has 56 streaming multiprocessors (SM), each limited to having up to 2048 threads, 32 thread blocks, and 64 KB shared memory [22]. For the bit-wise implementation of the  $V(D)J$  recombination algorithm with n-level granularity, each thread utilizes 48 registers, while there are 65536 registers available per SM. Therefore, the maximum number of active threads per SM is 1365 due to the register usage constraint. Also, it should be noted that the shared memory usage is not the limiting factor for the active threads per SM. As discussed in Section 3.2, the shared memory usage per block is 16 bytes plus one byte per thread for the counter value storage. Thus if we consider block size of 128 threads, only 134 bytes of shared memory is required per thread

block, allowing 489 thread blocks per SM. Given that for n-nucleotide length of nine, there are 2048 blocks for the 128 threads per block configuration. Due to register usage constraint, there are only 10 active thread blocks per SM. As a result, we do not reach the limiting factor (489 thread blocks per SM) for the shared memory usage.

### 3.5 Experimental Results

We start our analysis by determining the best thread-block configuration for different n-nucleotide lengths on a single GPU. We then compare the execution time of our bit-wise based implementation with the baseline [17] implementation for each n-nucleotide length. Finally, we present execution time analysis for the multi-GPU implementation with up to eight nodes.

#### 3.5.1 Thread Block Configuration Analysis

Fig. 3.3 shows the normalized results for four different thread block configurations over n-nucleotide length ranging from four to ten. For each length of n-nucleotide sequence, we take the shortest execution time and use that as a dividing factor over the execution time of other configurations. Therefore, normalized value of 1 represents the best performance for a given length. We did not consider n-nucleotide length of zero to three as there are not sufficient threads to utilize multiple *warps* executing concurrently. As shown in Fig. 3.3, there are negligible differences between performance of various thread block configurations for length of four to six since, there are not sufficient tasks to utilize all the available multiprocessors of the P100 GPU. For n-nucleotide length of less than seven, the thread utilization is below 14% as the total number of required threads is less than  $2^{14}$  while there are 114,688 threads available in P100. However, for n-nucleotide length of greater than seven, the workload increases such that more than 60% of available GPU threads are used. There is a 20% reduction in the performance for the n-nucleotide length of seven based on 256 threads per block configuration compared to other configurations. For n-nucleotide length of seven, the recombination pro-

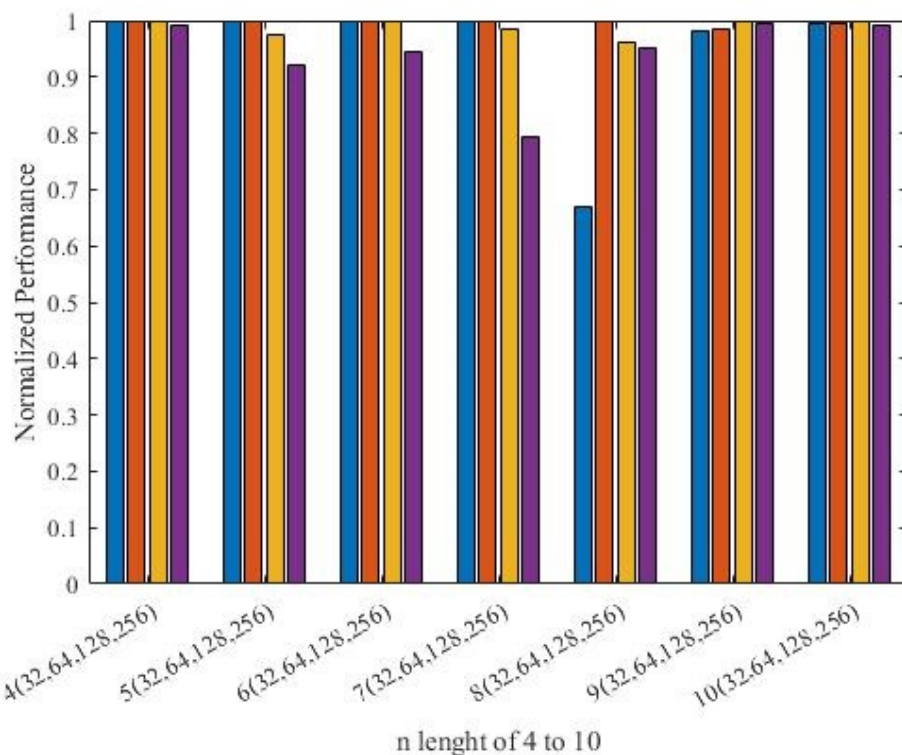


Figure 3.3: Normalized results of four different thread per block configurations (32, 64, 128, 256) for n-nucleotide length of four to nine.

cess completes in one iteration for every thread per block configuration since the total number of required threads is  $4^7 = 16,384$ , which is less than the total number of available threads in a single P100 GPU. The lower thread block utilization per SM is the root cause for this performance loss as shown in Table 3.4, which reports the thread, thread block, and *warp* utilization for each configuration.

We observe a 35% reduction in the performance for the n-nucleotide length of eight, if 32 threads per block are employed. The reason is that the maximum number of active thread-blocks per multiprocessor is 32 in the P100 GPU. Therefore, we are limited by the hardware to have 32 active blocks per SM in which each block has 32 threads. As a result, we have  $2^{10} \times 56$  active threads in GPU while we need  $2^{16}$  threads to complete the recombination process in one iteration. Let's consider the 64 threads per block configuration, based on the register constraint usage, we can have maximum 1365 active threads per



Table 3.4: GPU resource utilization for n-nucleotide length of seven with four different thread block configurations.

<i>thread block configuration</i>	<i>threads per SM</i>	<i>thread blocks per SM</i>	<i>warp</i>
32	1024 (50%)	32 (100%)	32 (50%)
64	1344 (65%)	21 (65%)	42 (65%)
128	1280 (62.5%)	10 (31.25%)	40 (62.5%)
256	1280 (62.5%)	5 (15.62%)	40 (62.5%)

SM, and based on the thread block configuration, we can have maximum of 21 blocks with 64 threads. This results in total of  $21 \times 64 \times 56 = 75,264$  threads, which is greater than the required threads for n-nucleotide length of eight. Therefore, the recombination process completes in one iteration for thread block configuration of 64 for n-nucleotide length of eight, while it can not be completed in one iteration with 32 threads per block. We note that the difference between the performance of 64, 128, and 256 threads per block configurations is negligible with normalized values of 1, 0.961, and 0.95 respectively as the recombination process is completed in one iteration for all three configurations.

For n-nucleotide length of nine, the total number of required threads is  $4^9 = 262,144$ , which is greater than available threads in a single GPU. This will results in completing the recombination process in more than one iterations. For 64, 128 and 256 threads per block configurations, four iterations is required to complete the recombination process. As a result, there is a negligible difference between their performances with normalized values of 0.984, 1 and 0.994 respectively. However, the number of required iteration increases by one, if 32 threads are employed per block. Therefore, the poorest performance belongs to 32 thread per block configuration for n-nucleotide length of nine.

In summary, based on Fig. 3.3, we set thread block configuration to 64 for n-nucleotide lengths four to eight, 128 for lengths nine and ten. In the following subsection, we evaluate the performance of bit-wise and multi-GPU implementations with respect to the the baseline implementation.

### 3.5.2 Bit-wise Simulation Results

In order to evaluate the bit-wise implementation, we ran an experiment on a single Tesla P100 GPU using the baseline implementation. The timing analysis and memory footprint for this experiment are used as a reference point for performance comparison.

Table 3.5 shows the total amount of required memory for  $V$ ,  $D$ ,  $J$  genes, and *in vivo* data set using the bit-wise representation. As stated in table 3.5, the memory footprint for constant memory reduces by a factor of 3.4 compared to the baseline implementation, while the required global memory reduces by a factor of 4. Table 3.6 shows the execution time results for each n-nucleotide length. Last row shows the total execution time for the recombination process. As shown, the total execution time reduces by a factor of 2.1 in comparison with the baseline implementation. For both implementations, after n-nucleotide of eight, the execution time increases by about a factor of four at each increments of n-nucleotide length by one. For n-nucleotide length of eight, we utilize 87.5% of available SM on a single GPU since 49 MPs with thread block configuration of 64 (smallest execution time) are used. Therefore, increasing the workload beyond this point directly results in increasing the execution time. The workload per GPU depends on the total number of unique n-nucleotide sequence as mentioned in 3.1. Therefore, increasing the length of n-nucleotide by one results in increasing the total number of unique n-nucleotide sequences by a factor of four and as a result execution time increases with the same factor.

### 3.5.3 Multi-GPU Simulation Results

Table 3.7 shows the execution time of the multi-GPU version of the bit-wise implementation for each n-nucleotide length. We ran experiments by using up to eight GPUs to evaluate the trends in execution time improvement with respect to change in number of GPUs.

The key observation from Table 3.7 is that there is slight increase in execution time if multiple GPUs are utilized for n-nucleotide length less than

Table 3.5: The memory footprint for the bit-wise implementation in comparison with the baseline approach.

<i>Gene</i>	<i>Baseline [17]</i>	<i>Bit-wise</i>	<i>Percentage reduction</i>
<i>V</i>	1448	425	70.65
<i>J</i>	3107	913	70.61
<i>D</i>	3210	908	71.71
<i>in vivo</i>	6517568	1629392	75.00

Table 3.6: Execution time on single GPU: Baseline vs. Bit-wise Implementations

<i>N length</i>	<i>Baseline (min)</i>	<i>Bit-wise (min)</i>
0	8.36	8.68
1	10.17	9.34
2	12.57	10.14
3	15.38	10.92
4	18.47	11.74
5	21.73	12.56
6	25.67	13.69
7	32.09	16.23
8	102.03	49.82
9	426.9	196.76
10	1755.35	797.8
Total	2428.7	1137.7

eight. The reason behind this observation is the fact that the P100 GPU is over-provisioned; the total number of required threads for any n-nucleotide length less than eight are less than the maximum  $2048 \times 56 = 57,344$  active threads. Moreover, the extra reduction step for a multi-GPU implementation becomes a slight overhead.

However, for n-nucleotide length more than seven, we observe a reduction in the execution time with multiple GPUs. This is due to the fact that a single GPU is almost fully utilized at 87.5% as explained in Section 3.5.1 for

Table 3.7: Execution Time for the bit-wise implementation on different number of GPUs.

<i>N length</i>	<i>1-GPU (min)</i>	<i>2-GPUs (min)</i>	<i>3-GPUs (min)</i>	<i>4-GPUs (min)</i>	<i>5-GPUs (min)</i>	<i>6-GPUs (min)</i>	<i>7-GPUs (min)</i>	<i>8-GPUs (min)</i>
0	8.68	9.23	9.28	9.09	9.09	9.09	9.1	9.09
1	9.34	9.91	9.97	9.77	9.77	9.78	9.78	9.78
2	10.14	10.69	10.75	10.55	10.56	10.56	10.55	10.56
3	10.92	11.48	11.55	11.34	11.35	11.35	11.35	11.35
4	11.74	12.29	12.37	12.12	12.13	12.12	12.13	12.12
5	12.56	13.18	13.24	13.03	13.03	13.01	13.01	12.97
6	13.69	14.34	14.39	13.91	13.90	13.90	13.89	13.88
7	16.23	16.61	15.66	15.43	15.32	15.28	15.26	15.23
8	49.82	27.89	22.77	18.92	17.62	17.57	17.55	17.52
9	196.76	112.86	82.00	58.60	46.59	40.22	34.36	28.70
10	797.8	455.73	301.18	231.37	185.31	159.48	139.62	116.22
Total	1137.7	694.21	513.16	404.13	344.67	312.36	286.6	257.42

n-nucleotide length of more than seven. Since the required number of threads exceeds the active thread count per GPU, we observe the benefit of the multi-GPU implementation for n-nucleotide length eight and above. At this point, we expect to see relatively linear reduction in the execution time for a given n-nucleotide length as we increase the number GPUs. However, the simulation results show a saturating execution time trend where adding another GPU resource no longer helps reduce the execution time. We further investigate this behavior in the following paragraph.

For n-nucleotide length of nine, the required number of threads is  $4^9 = 262,144$ , which is more than the available threads in a single P100 GPU. Based on the register resource constraint, the recombination process can be completed in four iterations ( $\lceil (4^9/1280 \times 56) \rceil = 4$ ) using a single GPU. In this case there are 47,104 active threads in the last iteration utilizing 65% of the GPU threads. Employing two GPUs results in completing the process in two iterations, while there are 32,768 threads in the last iteration. In this case we are only utilizing 45% of the GPU threads. Therefore, we do not observe two times speed up with two GPUs. Utilizing four GPUs for n-nucleotide length of nine results with completing the process in one iteration. Beyond this point, increasing the number of GPUs causes under-utilization of each GPU and does not significantly improve the execution time.

For n-nucleotide length of ten, the required number of threads is  $4^{10} = 1,048,576$ . The recombination process is completed in 15 iterations ( $\lceil (4^{10}/1280 \times 56) \rceil = 15$ ) using a single GPU while there are 45056 active threads in the last iteration (62% GPU threads utilization). However, employing two GPUs results in completing the process in 8 iterations. The reduction of execution time from 797 minutes to 455 minutes is proportional to the reduction of iteration count from 15 to 8, which is the root cause for not observing a linear speedup with two GPUs. Using three GPUs results in completing the recombination process in 5 iterations while the GPU thread utilization is at 87.6% in the last iteration. When we employ four GPUs, iteration count becomes 4. For the GPU count of five, six, and seven, the iteration count remains at 3 with fewer threads being utilized as the number of GPUs

increases. In order to complete the process in one iteration, we need to employ 16 GPUs. In overall, the saturation in the reduction of iteration count as we increase the GPU resources combined with the under utilization of the threads during the last iteration of the recombination process are the two root causes of saturating trend in execution time with respect to GPU count.

For the single GPU version, in the previous section, we showed that execution time increased by about a factor of four at each increments of the n-nucleotide length. Since we distribute the workload equally across the GPUs, we observe a similar trend for the multi-GPU implementation. For example, as shown in Table 3.7 execution time using two GPUs for n-nucleotide length of nine is about four times the execution time for n-nucleotide length of eight. Consistently we observe about a factor of four as we increase length from nine to ten for all GPU configurations.

Furthermore, we should expect the same execution time for two consecutive n-nucleotide lengths, while using one GPU for the first one and using four GPUs for the second one. As highlighted in Table 3.7, the execution time for n-nucleotide length of nine is 196 minutes by using single GPU. However we observe that execution time is 231 minutes for n-nucleotide of ten with four GPUs. We identify factors to this discrepancy as overhead of reduction process with the increase in number of GPUs, difference between the total number of  $nDn'$  combinations, and difference between the number of times each thread finds a match or terminates early. As stated earlier in Chapter 2, the  $D$  sequence can cut n-nucleotide sequence at any position, and each thread needs to generate all possible combinations of  $D$  with n-nucleotide sequence. As the length of n-nucleotide increases, the total possible combinations of  $nDn'$  increases by one for given  $D$  and n-nucleotide sequences. We note that an extra sequence needs to be combined with all possible forms of  $V$  and  $J$  gene sequences, which explains the difference between what we expected and what we observed. The difference in execution time reduces to around 9 minutes between n-nucleotide length eight with a single GPU and n-nucleotide length of nine with 4 GPUs. This discrepancy is about 2.6 minutes for the length pair of seven and eight with 1 and 4 GPUs respectively. We attribute this

discrepancy reduction trend to the three factors listed above.

## CHAPTER 4

### FPGA-based Implementation of The DNA Recombination Algorithm

In this Chapter, we study the implementation of  $V(D)J$  recombination process on FPGA using the n-nucleotide level parallelism that was used in the GPU-based implementation in Section 4.1. As we converge to a final hardware architecture, we provide a detailed explanation for each unit individually. We explain the experimental results for the n-level FPGA-based implementation of  $V(D)J$  recombination process in Section 4.2. Based on the simulation results, we explain the draw back of n-level parallelization method for FPGA-based implementation of the recombination process in Section 4.3. As a result, we proposed the VJ level parallelization approach for the FPGA-based implementation to overcome the draw back of n-level approach in Section 4.4. We describe the structure of each unit of the final hardware architecture for the VJ method. Finally, we describe the simulation results for the new parallelization approach in Section 4.5.

#### 4.1 Hardware Implementation of N Level Parallelization

The goal for the hardware implementation of DNA recombination process is to accelerate the generation of all possible TCR sequences with any given  $V$ ,  $D$  and  $J$  gene sequences and count the number of times each sequence can be generated artificially. As we converge to a final hardware architecture for this process we need to determine the level of parallelism and granularity of the processing elements for generating *in silico* sequences, orchestrate the data transfers between the memory and computation units, and balance the trade-off between throughput and resource usage. In this section, we first describe the parallelization strategy and then explain the structure of each unit that is used to implement the  $V(D)J$  recombination algorithm.

Fig 4.1 shows the proposed architecture for the hardware implementation



of the recombination process, which consists of three units, address generator unit (AGU), memory bank unit, and processing unit. The processing unit (PU) consists of processing element (PE), n-sequence initiator, and *in vivo* address generator units. The PE consists of a number of parallel computation units (CUs), which is formed of concatenation and comparison units.

The PU generates the number of times each *in vivo* sequence can be generated artificially in cooperation with the memory bank unit and corresponds to the inner most three loops in Algorithm 1. The n-sequence initiator provides the legitimate n-nucleotide length and reference n-nucleotide sequence for each CU based on the length of each input  $V$ ,  $D$  and  $J$  gene sequences. Each CU generates a unique n-nucleotide sequence, forms all possible *in silico* sequences based on the input  $V$ ,  $D$  and  $J$  gene sequences, and searches those generated sequences in the *in vivo* memory bank. The *in vivo* address generator unit allows reading the *in vivo* sequences and their counter values for each CU, and updates the counter value when a match is found by a CU. The AGU is corresponds to the outer most three for loops in Algorithm 1 in order to generate the indexes for accessing the  $V$ ,  $D$ ,  $J$  sequences. The memory bank unit is compromised of four sub-memory banks for maintaining input and output data sets.

#### 4.1.1 N Level Parallelization Strategy

In this section, we present a series of experimental analysis to answer the following four questions that will help us determine the degree of parallelism ( $D_p$ ) to realize in the final architecture:

- 1) What is the efficient parallelization strategy that will minimize the execution time, 2) How does the workload partitioning strategy across the CUs affect the performance based on the resource constraints imposed by the target FPGA, 3) What is the correlation of the  $D_p$  with the resource utilization and the critical path delay, and 4) How does the  $D_p$  affect the throughput?

As mentioned in Chapter 2, all possible forms of n-nucleotide sequences are involved in the recombination process. The length of n-nucleotide ( $N_{length}$ ) can be between zero to ten and the n-nucleotide sequence can be placed on either

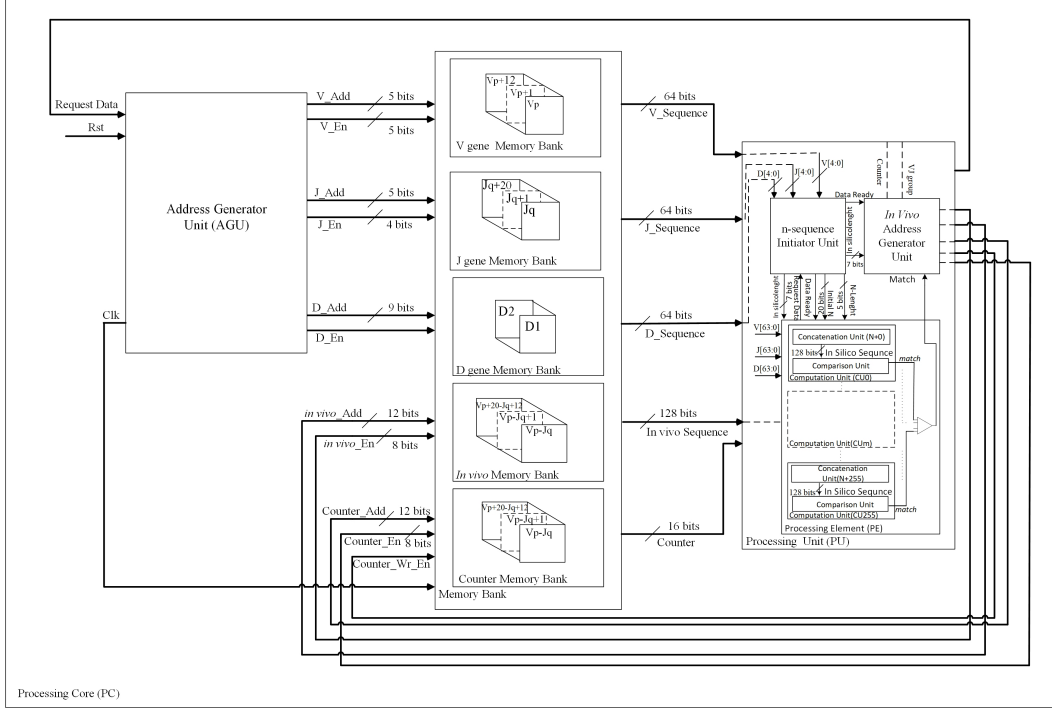


Figure 4.1: The hardware implementation of  $V(D)J$  recombination process using the N-level parallelization approach.

side of  $D$  sequence. Table 4.1 shows the total number of unique  $n$ -nucleotide sequences based on the  $N_{length}$  along with the total number of possible combinations of  $n_1 D n_2$  with a given  $D$  sequence given that  $D$  gene can partition the  $n$ -nucleotide of size  $N_{length}$  at any position generating  $N_{length} + 1$  such partitioning opportunities. Therefore, if the length of  $D$  sequence is greater than zero then, the total possible combinations is equal to  $4^{N_{length}} \times (N_{length} + 1)$ .

The computation requirement is the same for generating the *in silico* sequences based on each of the  $4^{N_{length}}$  unique  $n$ -nucleotide sequences. This creates an opportunity to parallelize at  $n$ -nucleotide level and achieve a balanced workload distribution across the computation units. This was the parallelization approach taken in our GPU-based implementation to achieve a balance workload across the threads and thread blocks of the GPU. Consider the case of  $N_{length}$  of ten, which generates over one million unique sequences. Hardware implementation at a scale of million CUs is not feasible due to the resource constraints. Therefore a partial parallelization is needed and the resource de-

mand for a single CU helps us determine the amount of unrolling we can apply to the loop with index four in Algorithm 1 and derive the number of CUs we can instantiate on the target FPGA. This number will also help us determine the structure of the  $n$ -sequence initiator and PU. Therefore, in order to answer the second question, we implement a single CU, that is composed of concatenation and comparison units on the the Virtex-7 XC7VX485T FPGA. We will describe the details of the PU structure in section 4.1.2. As shown in table 4.6, the "Slice LUTs" is the determining factor and we can fit up to 256 CUs leaving 15% of the resources for the AGU state machine,  $n$ -sequence initiator unit, *in vivo* address generator and glue logic.

In order to answer the third question, we implement the proposed architecture for different  $D_p$  and plot the resource utilization trend as shown in Fig. 4.2. As we increase the  $D_p$ , the resource utilization increases linearly. We show the critical path delay with respect to changes in  $D_p$  in Fig. 4.3 to answer the third question. As shown, the critical path delay slightly increases as the  $D_p$  increase.

In order to answer the last question, we first need to define the throughput. The throughput is defined as the total number of comparisons per second. Since, we have the same number of comparison units as the  $D_P$ , the throughput increases linearly.

Based on the above analysis, we set the  $D_P$  equal to 256 and the final architecture based on our  $n$ -nucleotide level parallelization strategy is shown in Fig. 4.1. In the following sections, we explain the structure of each unit within the processing core (PC) in detail.

#### 4.1.2 Processing Unit

As illustrated in Algorithm 1, modeling the TCRs repertoire involves six nested for loops. We map the inner most three for loops with index 4-6 to the PU to iterate through all possible forms of  $n$ -nucleotide additions, create *in silico* sequences, and search for a match of the *in silico* sequence within the *in vivo* data set. We describe the structure of the PU composed of the PE,  $n$ -sequence initiator, and *in vivo* address generator units in the following subsections.

Table 4.1: The total number of unique  $n$ -nucleotide sequences and total number of possible combinations of  $n_1Dn_2$  with a given  $D$  sequence based on the length of  $n$ -nucleotide.

$N_{length}$	<b>Total number of unique <math>n</math> – nucleotide sequences</b>	<b>Total number of possible combinations of <math>n_1Dn_2</math> with a given <math>D</math></b>
0	1	4
1	4	8
2	16	48
3	64	256
4	256	1280
5	1024	6144
6	4096	28672
7	16384	131072
8	65536	589824
9	262144	2621440
10	1048576	11534336

Table 4.2: Resource Utilization for a single CU

Slice LUTs (303600)	977
Slice Registers (607200)	509
Slice (75900)	275
hline LUT as Logic (303600)	977
LUT Flip Flop Pairs (303600)	272

The inputs to the PU are the  $V$ ,  $D$ ,  $J$ , and *in vivo* sequences received from their respective memory bank units, the counter received from the counter memory bank unit, and the VJ group ( $group_{id}$ ) from the AGU. The addresses for these three sequences are determined by the AGU, which will be discussed later. We set the bitwidth for each sequence to 64 as shown in Fig. 4.1. In C57BL/6 mice data set, the maximum length of the  $V$ ,  $J$  and  $D$  gene sequences is 26 characters. We assign two bits to represent each of the four types of nucleotide bases (A, G, C, and T). We reserve five bits to keep track of the length of each sequence. The  $n$ -sequence initiator unit will rely on this length information for selecting valid length for the  $n$ -nucleotide sequence.

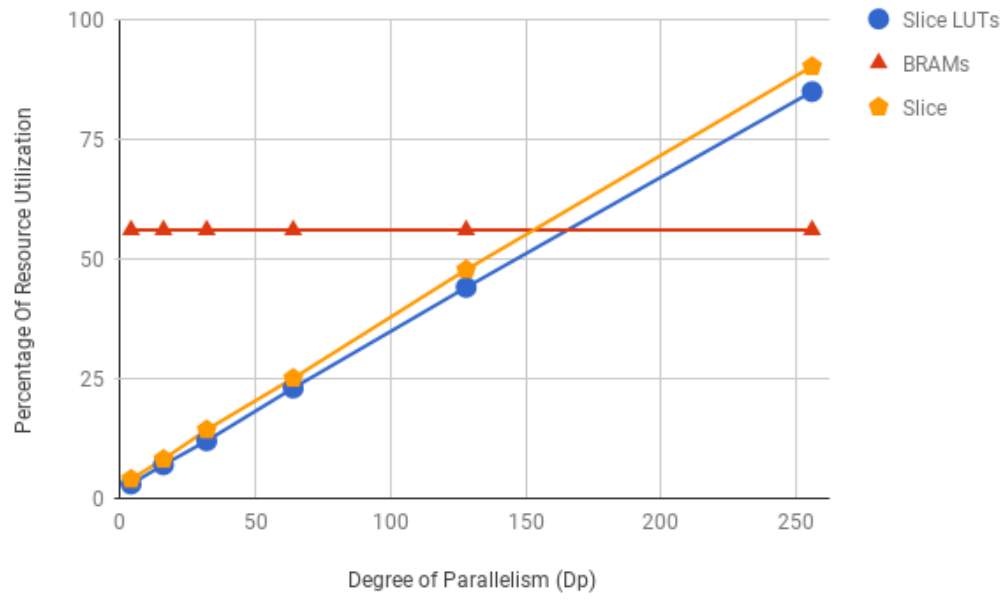


Figure 4.2: The resource utilization of the proposed architecture for different values of the  $D_P$ .

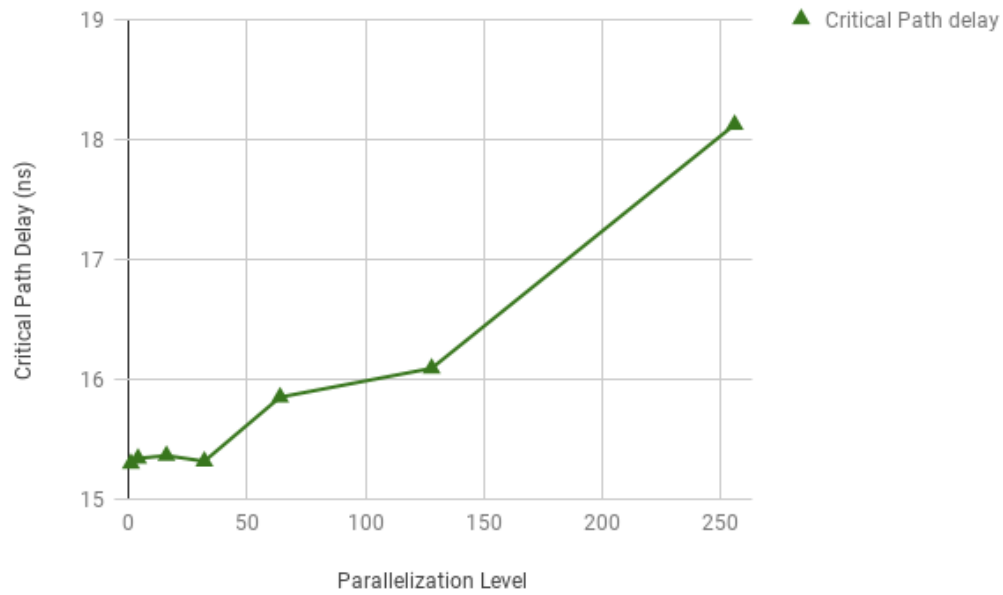


Figure 4.3: The critical path delay of the proposed architecture for different values of the  $D_P$ .

In addition, the concatenation unit depends on the length information for generating the *in silico* sequence. As explained earlier in Chapter 2, the  $D$  gene goes through chewback process on its both ends. Based on the C57BL/6 mice data set, there are maximum of 22 different paths to generate  $D$  sequences. We reserve five bits to represent each path, which we refer to as  $D_{Path}$ . The *in vivo* address generator unit will require this information for counter update, which will be discussed later. Therefore, in total we need to form a 62 bit package as an input to the PU, however we set the package size as 64 leaving 2-bit for debugging purpose to set as used or not used.

### 4.1.3 n-sequence Initiator Unit

The input to the n-sequence initiator unit are the length of  $V$ ,  $D$ , and  $J$  gene sequences received from their respective memory bank units, the *RequestData* signal received from the PE. The outputs of this unit are the *DataReady*,  $N_{length}$ , length of *in silico* sequence ( $sequence_{length}$ ), and the reference n-nucleotide sequence ( $Initial_N$ ) for the CUs, and *DataReady* and  $sequence_{length}$  for the *in vivo* address generator unit. We set the bitwidth of  $N_{length}$  and  $Initial_N$  to five and 20 respectively, as the maximum length of n-nucleotide is 20 bits (ten characters). Also, we set the bitwidth of  $sequence_{length}$  to seven as the maximum value for  $sequence_{length}$  is 120 bits (60 characters).

The n-sequence initiator unit implements the for loop indexed as four in the Algorithm 1. This loop iterates through n-nucleotide of length zero through ten and generates all possible sequences for that length. We can extend the for loop with index of four into three sub-for loops as shown in Algorithm 3.

The first for loop iterates through all possible n-nucleotide sequences of  $N_{length}$  between zero and ten. However, all possible n-nucleotide sequences for each length will not result with valid *in silico* sequence based on the given  $V$ ,  $D$  and  $J$  gene sequences as the  $sequence_{length}$  must be within the range of six to sixty characters and divisible by three. Therefore, primary functionality of the n-sequence initiator unit is selecting the legitimate  $N_{length}$  and proceed to the next loop. The second for loop iterates through all possible unique n-nucleotide sequences for the selected  $N_{length}$ . Table 4.1 shows the total number

---

**Algorithm 3:** Pseudo code for expanding the for loop with index of four in Algorithm 1.

---

```

1 for  $i = 0$  to 10 do
     $N_{length} = i$ ;
     $sequence_{length} = V_{len} + J_{len} + D_{len} + N_{length}$ ;
    if ( $6 \leq sequence_{length} \leq 60$ ) & ( $sequence_{length} \% 3 == 0$ ) then
2         for  $k = 0$  to  $4^i$  do
3              $Initial_N = k$ ;
            for  $j = i$  to 0 do
                 $len_{n_1} = j$ ;
                 $len_{n_2} = i - j$ ;
                 $k = k + D_P$ ;

```

---

possible unique n-nucleotide sequences with different length.

Rather than having this unit generate a n-nucleotide sequence for each CU, and send all possible combinations in rounds each with 256 sequences (as there are 256 parallel CUs), we send only  $Initial_N$  as a reference starting address to all CUs and have each CU calculate its unique n-nucleotide sequence using  $Initial_N + CU_{id}$ , expression where  $CU_{id}$  is the index of CU. This allows us to reduce the amount of data transfer to the CUs and wiring requirement on the implementation. After each round, the  $Initial_N$  is incremented by 256 to complete all the rounds needed for that specific  $N_{length}$ , which is  $4^{N_{length}}/256$ . The third for loop iterates through all possible combinations for the length of  $n_1$  and  $n_2$  sequences to cover all forms of  $n_1 D n_2$  sequence.

#### 4.1.4 In vivo Address Generator Unit

The inputs for the *in vivo* address generator unit are the  $sequence_{length}$  and  $DataReady$  from the n-sequence initiator, the  $group_{id}$  from the AGU, the  $Match$  signal indicates that there is a match between *in vivo* and *in silico* sequences, from the  $PE$ , and the 16-bit counter indicating the number of times that the specific *in vivo* sequence is generated, from the counter memory bank. The outputs are the address for reading the sequence from the *in vivo* memory bank ( $in\ vivo\_Add$ ) along with its read enable signal

(*in vivo\_En*) and the address for the sequence's counter value in the counter memory bank (*Counter\_Add*) along with read enable (*Counter\_En*) and write enable (*Counter\_Wr\_En*) signals to control reads and writes. We set the bitwidth of *Counter\_Add* and *in vivo\_Add* to 12 as there are maximum 3249 *in vivo* sequences in the BRAMs. In addition, we set the bitwidth of *in vivo\_En* and *Counter\_En* to 8 as there are 240 BRAMs in the *in vivo* and counter memory bank units.

Before we describe the functionality of the *in vivo* address generator unit, we need to explain the structure of the *in vivo* memory bank. As mentioned in Section 2.0.3, the *in vivo* sequences are grouped based on the *VJ* pair used to generate that sequence. We benefit from this feature to reduce the comparison search space. Therefore, we distribute the *in vivo* data set based on the *VJ* group assignment across 240 BRAMs, where each BRAM holds *in vivo* sequences sorted in ascending order of length for that *VJ* group. This way of data organization allows us to use the *group\_id* to search for the data only in one of the 240 BRAMs, and *sequence\_length* to search within a single BRAM among the sequences that have the same length. In order to realize this, we need to know the starting and ending addresses of the *in vivo* sequences in the *in vivo* memory bank for any given *sequence\_length*. Therefore, we utilize two 240x19 arrays (*starting\_array*, *ending\_array*), which keep the starting and ending address of *in vivo* sequences for each *sequence\_length* and *group\_id*. The size of *starting\_array*, *ending\_array* is defined based on the total number of *VJ* groups (240 *VJ* groups) and the total number of available length for the *in vivo* sequence (19 different lengths from six to sixty characters).

#### 4.1.5 Processing Element

The inputs for the PE are the *V*, *D* and *J* gene sequences from the memory banks, the *sequence\_length*, *N\_length*, *DataReady* and *Initial<sub>N</sub>* from the n-sequence initiator unit and 128-bit *in vivo* sequence from the *in vivo* memory bank unit. The output is the *Match* signal for the *in vivo* address generator unit indicating that one of the 256 CUs generated the *in vivo* sequence. This *Match* signal triggers the *in vivo* AGU to update the corresponding counter



for that *in vivo* sequence. As shown in Fig. 4.1, the PE consists of 256 parallel CUs where, each CU consists of a pair of concatenation and comparison units.

The concatenation unit uses an *or* gate to concatenate the  $V$ ,  $n_1$ ,  $D$ ,  $n_2$  and  $J$  sequences in the given order to form a 128-bit sequence. The position of each sequence in the 128-bit package is determined by the length of the remaining four sequences. This position value is used as the shift amount to apply to each sequence before the *or* gate based concatenation. We pass the *in silico* sequence to the comparison unit of each CU to compare the generated sequence with the *in vivo* sequence. The 128-bit comparator unit sets the *match* flag to one when the 128-bit *in silico* sequence is equal to the *in vivo* sequence, which in turn triggers the *Match* signal.

#### 4.1.6 Address Generator Unit

The main objective of the AGU is to generate the addresses for the  $V$ ,  $D$  and  $J$  memory bank units. The input for the AGU is the one bit *RequestData* signal received from the PU. The outputs are the addresses for reading three sequences from the  $V$ ,  $J$ ,  $D$  memory banks ( $V\_Add$ ,  $J\_Add$ ,  $D\_Add$ ) along with their read enable signals ( $V\_En$ ,  $J\_En$ ,  $D\_En$ ). We set the bitwidth of  $V\_Add$  and  $J\_Add$  to five as there are maximum 26 sequences in their BRAM. In addition, we set the bitwidth of  $D\_Add$  to nine as there are maximum 202  $D$  gene sequences in its BRAM. The bitwidth of  $V\_En$  is set to five as there are 20 BRAMs where, each BRAM holds one basic  $V$  gene sequence and its chewback and palindromic forms 2.0.3. Also, we set the bitwidth of  $J\_En$  to four as there are 12 basic  $J$  genes where, each gene and its chewback and palindromic forms are stored in one BRAM.

## 4.2 Simulation Results of the N Level Parallelization Architecture

Table 4.3 shows the resource utilization for the n level parallelization architecture. As shown, we utilize up to the 85% of the available resources on the target FPGA. The critical path delay is 19.798 *ns* and the maximum clock rate is 50 MHz. For comparison purpose, we ran experiment for the bit-wise GPU-

based implementation using P100 GPU. Table 4.4 shows the execution time for each  $N_{Length}$  on both Virtex-7 FPGA and P100 GPU. The total execution time is equal to 19571 minutes ( $\sim 13.6$  days) on the target FPGA while the total execution time is equal to 1137 minutes ( $\sim 19$  hours) on a single GPU.

### 4.3 Drawbacks of N Level Parallelization Architecture

As stated in table 4.4, the execution time for FPGA-based implementation using n-level parallelization strategy is 17 times slower than the bit-wise GPU-based implementation. This is due to the fact that the critical path delay is large for the FPGA-based implementation, which results in low clock rate frequency. In the following we state the advantages and disadvantages of n-level parallelization strategy for FPGA-based implementation.

One of the main advantages of n level parallelization architecture is the even workload distribution across the computation units since they work on the same  $V$  and  $J$  genes, their comparison search space is equal. The second advantage is the highly parallel comparison process. As shown in Fig. 4.1, there are 256 CUs, which compare their *in silico* sequence with the *in vivo* sequence. Indeed, the execution time for comparison process reduces by factor of 256 in comparison with the sequential based comparison process. The third advantage is the elimination of the repetitive computations for generating *in silico* sequences. The n-sequence initiator unit perform required computations such as determining the valid length for n-nucleotide and *in silico* sequences and pass them to the CUs. This result in performing computation once for 256 CUs.

As stated above, the main disadvantage of n-level parallelization is the large critical path delay, which results in low clock frequency rate. The large critical path delay is due to the communication between the n-sequence initiator unit and CUs. In order to remove the communication, we need to perform all required computations within the CU, which results in increasing the resource utilization and reducing the  $D_p$ . Another disadvantage of this architecture is that there is no overlap between the execution of two consecutive component. For example, every components are in idle while the comparison units compare

Table 4.3: FPGA Resource Utilization for the N Level Parallelization Architecture.

<i>Component</i>	<i>Slice LUTs</i>	<i>Slice Registers</i>	<i>Slice</i>	<i>F7 Muxes</i>	<i>F8 Muxes</i>	<i>LUT Flip Flop Pairs</i>	<i>BRAM</i>
	<i>303600</i>	<i>607200</i>	<i>75900</i>	<i>151800</i>	<i>75900</i>	<i>303600</i>	<i>1030</i>
<i>n</i> – sequence generator unit	1606	102	487	298	32	36	0
<i>in vivo</i> address generator unit	1352	181	838	2	0	181	19
PU	253103	129080	66564	0	0	69454	0
AGU	71	44	27	1	0	31	0
<i>V</i> memory bank unit	0	0	0	0	0	0	20
<i>J</i> memory bank unit	0	0	0	0	0	0	12
<i>D</i> memory bank unit	0	0	0	0	0	0	2
<i>in vivo</i> memory bank unit	0	0	0	0	0	0	452
<i>counter</i> memory bank unit	0	0	0	0	0	0	70
glue and display logics	1431	216	469	4	0	98	0
Total	257563	129623	68385	305	32	69800	575
Percent	84.84%	21.34%	90.09%	0.20%	0.042%	22.99%	56%

Table 4.4: Execution time for the FPGA and bit-wise GPU-based implementations using Virtex-7 and P100 GPU, respectively.

$N_{Length}$	<i>Execution Time (min) for FPGA-based implementation</i>	<i>Execution Time (min) for GPU-based implantation</i>
0	<1	8.68
1	1	9.34
2	1	10.14
3	1	10.92
4	1	11.74
5	8	12.56
6	39	13.69
7	180	16.23
8	805	49.82
9	1768	196.76
10	15243	797.8
Total	19571	1137.7

the *in silico* and *in vivo* sequences.

#### 4.4 Hardware Implementation of VJ Level Parallelization

In this section, we propose the VJ level parallelization approach for FPGA-based implementation of the  $V(D)J$  recombination process. We first analyze the implementation of VJ level parallelization approach for FPGA-based implementation and determine the final architecture in Section 4.4.1. Then, we provide a detailed explanation about the structure of each unit individually in Sections 4.4.2 - 4.4.4. Finally, we ran experiment to evaluate the performance of VJ level method and provide the experimental results in Section 4.5.

##### 4.4.1 VJ Level Parallelization Strategy

In this section, we present a series of experimental analysis to answer the following two questions that will helps us determine the parallelization level of the final architecture:

- 1) What is the resource requirement for a processing core (PC) that is

Table 4.5: Resource Utilization for a single PU

Slice LUTs (303600)	2007
Slice Registers (607200)	900
Slice (75900)	642
LUT as Logic (303600)	2007
LUT Flip Flop Pairs (303600)	735

composed of an address generator unit, memory banks, combination and comparison units?

2) What is the efficient data partitioning strategy across the PCs that will maximize the execution time performance, and how does the partitioning strategy affect the parallelization level (number of PCs), workload per PC, and execution time performance based on the resource constraints imposed by the target FPGA?

The resource demand of a single PU will help us determine the number of PCs we can instantiate on the target FPGA. This number will also help us determine the structure of the address generation unit (AGU) and the way the data should be partitioned across the block rams. Therefore, in order to answer the first question, we implement a single PU, that is composed of combination and comparison units along with a buffer in between these units on the Virtex-7 XC7VX485T FPGA. As shown in table 4.5, the "Slice LUTs" is the determining factor and we can fit up to 136 PUs leaving 10% of the resources for the AGU state machine and glue logic.

We need to consider the following two conditions for data partitioning strategy before answering the second question. First, the data partitioning strategy should result in even workload across the PCs. Second, the data distribution strategy should allow the PCs operate independently meaning that the reads from the memory banks when accessing the  $V$ ,  $D$ ,  $J$ , and *in vivo* sequences along with the writes to memory banks when updating the counter value should be isolated from each other. In the following, we explain the available options for the data distribution strategy. Note that these strategies were introduced based on the structure of input data set.

One solution would be to pair one  $V$  gene, one  $J$  gene and both  $D$  genes to each PU. Based on this assignment, the total number of required PCs will be 240. Table 4.5 shows that we can only realize up to 136 PUs on the target FPGA. Therefore, this solution will result in completing the recombination process in two rounds, in which during the second round 24% of the PUs would be idle.

We distribute the *in vivo* data set across the PUs based on the  $VJ$  group 2.0.3 assignment so that we satisfy the second condition. Fig. 3.1 illustrates the normalized distribution of *in vivo* data set across 240  $VJ$  pairs. Even though the workload for each combination unit across the PUs is homogeneous, this is not the case for the comparison unit since the total number of *in vivo* sequences is not distributed evenly across the  $VJ$  groups. However, this distribution allows us to exploit  $VJ$  level parallelism with independent and parallel memory reads with a trade off in uneven workload distribution for the comparison unit. Furthermore, since each *in vivo* sequence is originated from a specific  $VJ$  pair, the recombination paths that may generate the same sequence will always come from the same  $VJ$  pair. Therefore for each successful recombination, the counter updates will occur within the same PU ensuring that PCs will always write into their designated counter memory units.

Alternative solution would be increasing the workload per PUs till we reach a single iteration based execution. The strategy that allows us to complete the process in one iteration requires one  $V$  and two  $J$  genes per PU, resulting with a total of 120 PUs to cover for all 240 pairs. We can keep increasing the number of  $J$  genes per PUs, but such an approach has two major drawbacks. As we increase the number of  $J$  genes per PU we reduce the number of parallel PUs and increase the workload per PU, which result in increasing the execution time.

Fig. 4.4 shows the architecture of proposed PC based on the  $VJ$ -level parallelization strategy. In the following, we explain the structure of each unit within the PC in detail.

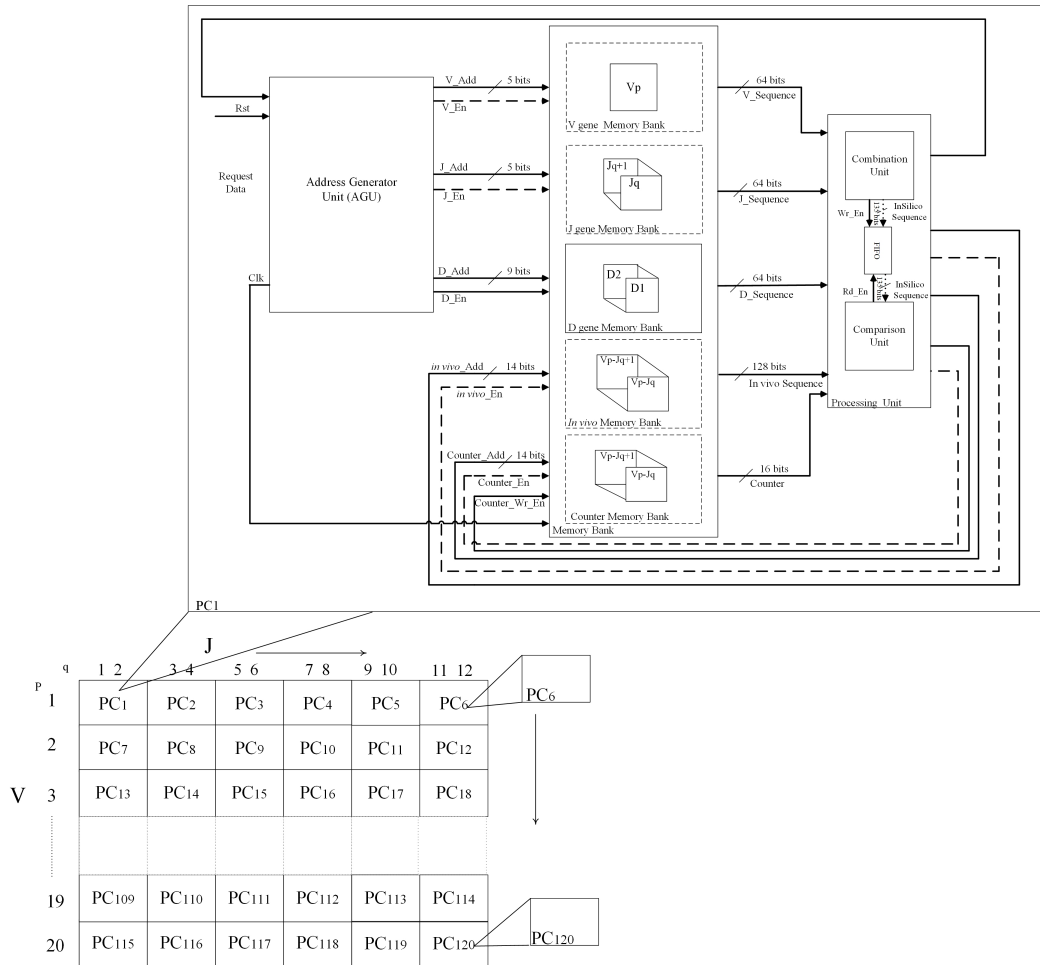


Figure 4.4: A high level view of hardware implementation of VJ level parallelization for the  $V(D)J$  recombination process.

#### 4.4.2 Combination Unit

The inputs to the combination unit are the  $V$ ,  $J$  and  $D$  gene sequences received from their respective memory banks. The addresses for these three sequences are determined by the AGU, which will be discussed later. We set the bitwidth for each sequence to 64 bits as shown in Fig. 4.4. In C57BL/6 mice data set, the maximum length of the  $V$ ,  $J$  and  $D$  gene sequences is 26 characters. We assign two bits to represent each of the four types of nucleotide bases (A, G, C, and T). We reserve six bits to keep track of the length of each sequence. The combination unit will rely on this length information for the *in silico* generation. As explained earlier in section 2, the  $D$  gene goes through chewback process on its both ends. Based on the C57BL/6 mice data set, there are maximum of 22 different paths to generate  $D$  sequences. We reserve five bits to represent each path, which we refer to as  $D_{Path}$ . The comparison unit will require this information for counter update, which will be discussed later. Therefore, in total we need form a 63 bit package as in input to the Combination Unit, however we set the package size as 64 leaving 1-bit for debugging purpose to set as used or not used. The combination unit continuously executes the following five steps as long as the buffer is not full to implement the for loops indexed four and five of the Algorithm 1. Fig. 4.5 shows the high level flow of these five steps.

*Step 1:* We set the length of n-nucleotide to zero as a starting point. For the current 64-bit  $V$ ,  $J$  and  $D$ , we extract the length of each sequence and calculate the length of *in silico* sequence using (4.1). Based on the C57BL/6 mice data set there are two conditions that need to be met in terms of the *in silico* sequence length. Length of the generated sequence must be within the range of six to sixty characters and divisible by three. If the length of *in silico* sequence meets those two conditions, then we set the initial binary value of the n-nucleotide sequence as zero for the current n-nucleotide length and proceed to the step two. If the length conditions are not met, then we check the length of n-nucleotide. If the length of n-nucleotide is less than ten then, we increase the length of n-nucleotide by one, update the length of *in silico* sequence, and reevaluate the new length. If the length of n-nucleotide is not



less than ten then, we request a new input data from the V, D, and J memory banks, whose addresses are generated by the AGU.

$$Length_{in\ silico} = V[5 : 0] + J[5 : 0] + D[5 : 0] + n - nucleotide_{length} \quad (4.1)$$

*Step 2:* The n-nucleotide sequence involves the  $n_1$  and  $n_2$  nucleotide sequences 2.0.1. In the second step, we set the length of  $n_1$  and  $n_2$  sequences based on the length of n-nucleotide, which was selected in the previous step. We set the length of  $n_1$  sequence equal to the length of n-nucleotide and set the length of  $n_2$  sequence equal to zero, then we continue on to the third step. Later in step five we will decrement and increment the  $n_1$  and  $n_2$  lengths by one respectively till  $n_1$  is 0 and  $n_2$  is the length of n-nucleotide.

*Step 3:* In this step, we use an *or* gate to concatenate the V,  $n_1$  D,  $n_2$  and J sequences in the given order to form a 134-bit sequence. The position of each sequence in the 134-bit package is determined by the length of the remaining four sequences. This position value is used as the shift amount to apply to each sequence before the *or* gate based concatenation. As mentioned before, the maximum length of an *in silico* sequence is 60 characters (120 bits). We reserve six bits to keep track of the length of *in silico* sequence ( $sequence_{length}$ ), two bits to keep track of the VJ group  $group_{id}$ , five bits to indicate the  $D_{Path}$ , and one bit as a  $n_{length}$  flag to show that the *in silico* sequence is created using the n-nucleotide with length of greater than zero. The comparison unit relies on the *in silico* length information and VJ group to reduce its search space, which we will explain later.

*Step 4:* In this step, we check the status of the the first-in, first-out buffer (FIFO). If the FIFO is not full, then we pass the generated 134-bit sequence into the buffer and proceed to the fifth step. Otherwise, we remain in this step till the last entry in the buffer becomes available.

*Step 5:* At this point of the execution, the combination unit has generated an *in silico* sequence and fed it into the FIFO. This step adjusts the lengths of  $n_1$  and  $n_2$  sequences along with the binary value of the n-sequence based on the following execution model. We first evaluate the length of  $n_1$  sequence.

If it is greater than zero, then we increase the length of  $n_2$  sequence, decrease the length of  $n_1$  sequences, and proceed to the third step. We increase and decrease the length  $n_1$  and  $n_2$  respectively to generate all possible combinations of  $n_1 D n_2$  sequence.

When the length of  $n_1$  sequence reaches to zero, we compare the current binary value of the  $n$ -nucleotide sequence with the largest number that can be generated for that length (note that length was determined in Step 1). If it does not match, then we increase the  $n$ -nucleotide value by one and move to the second step. Otherwise, we move to the first step to evaluate and increase the length of  $n$ -nucleotide. Indeed, all possible forms of  $n$ -nucleotide sequence can be generated by increasing its value by one. Since all zeros in the  $n$ -nucleotide sequence is equivalent to the sequence with all 'A', while the  $n$ -nucleotide sequence with all ones is equal all 'T'.

#### 4.4.3 Comparison Unit

The inputs for the comparison unit are 134-bit package received from the buffer, 128-bit *in vivo* sequence received from the *in vivo* memory bank, and the 16-bit counter indicating the number of times that specific *in vivo* sequence has been generated. Before we describe the functionality of the comparison unit, we explain the structure of the *in vivo* memory bank. As mentioned in section 4.4.1, we distribute the *in vivo* data set across the PCs based on the  $VJ$  group assignment. Since we pair each  $V$  gene with two  $J$  genes, we divide the *in vivo* memory bank into two partitions, where each partition holds *in vivo* sequences sorted in ascending order of length for that  $VJ$  group. This way of data organization allows us to use the  $group_{id}$  to search for the data only in one of the two partitions, and  $sequence_{length}$  to search within a single group among the sequences that have the same length. In order to realize this, we need to know the starting and ending addresses of the *in vivo* sequences in the *in vivo* memory bank for any given  $sequence_{length}$  and  $group_{id}$ . Therefore, we utilize two 2x19 arrays ( $starting_{array}$ ,  $ending_{array}$ ), which keep the starting and ending address of *in vivo* sequences for every  $sequence_{length}$  and  $group_{id}$ . The size of  $starting_{array}$ ,  $ending_{array}$  is defined based on the total number of

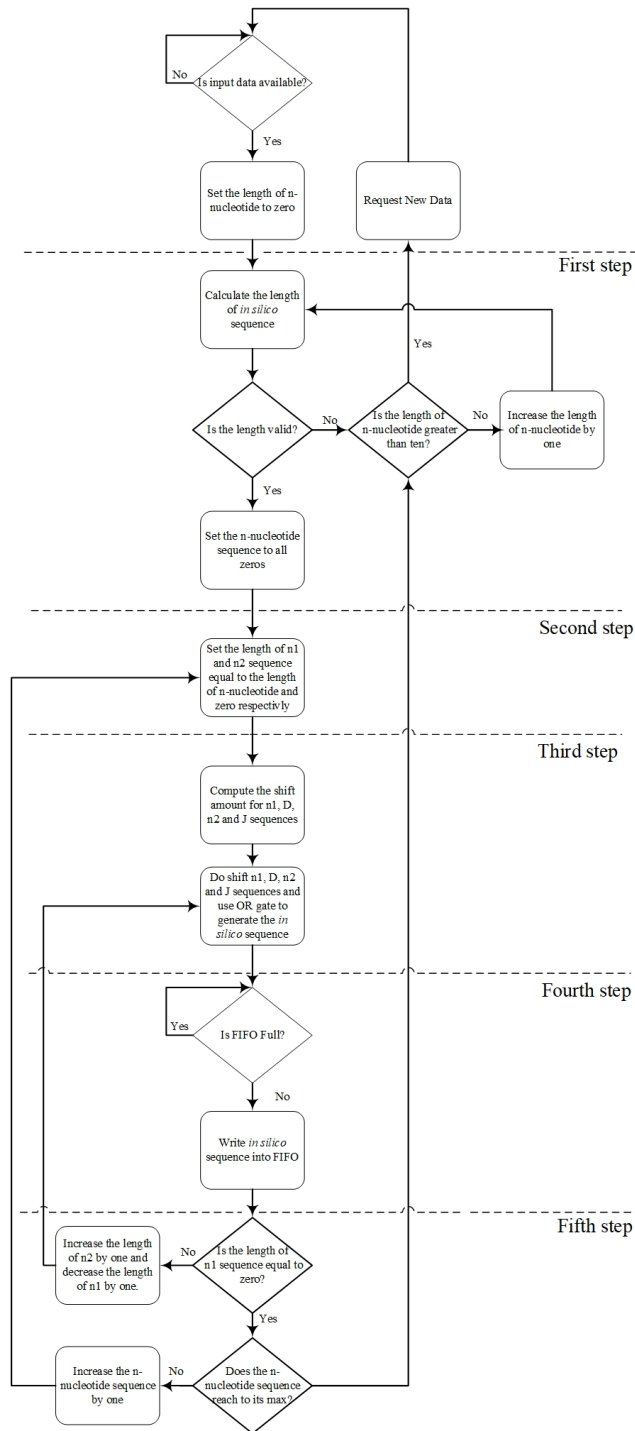


Figure 4.5: The algorithmic description for the structure of Combination unit.

$VJ$  group in each PC (two  $VJ$  groups) and the total number of available length for the *in vivo* sequence (19 different lengths from six to sixty characters).

The comparison unit continuously executes the following four steps as long as there is data in the buffer to implement the inner most for loop of the Algorithm 1. Fig. 4.6 shows the high level flow of these four steps.

*Step 1:* In this step, we check whether the buffer has at least four entries or not. We remain in this step till the combination unit passes at least four *in silico* sequences to the buffer.

As mentioned before, there are four nucleotide bases (A, C, G, and T) that we need to use for generating the n-nucleotide sequences. Therefore, there are  $4^M$  n-nucleotide sequences with length of  $M$  involving in the recombination process. In addition, we need to generate all possible combinations of  $n_1 D n_2$  in the recombination process (step five of the combination unit) for any given n-nucleotide sequence in the presence of  $D$  sequence ( $D$  sequence with length of greater than zero). This causes additional  $M + 1$  ways for generating the *in silico* sequence for a given n-nucleotide. Therefore, the total number of possible *in silico* sequences for a given  $V$ ,  $D$ , and  $J$  genes is equal to  $4^M \times (M + 1)$  using the n-nucleotide sequence with length of  $M$ . Based on this analysis, we have at least  $4^M \times (M + 1)$  *in silico* sequences with the same features (*sequence<sub>length</sub>*, *group<sub>id</sub>*, and *D<sub>path</sub>*) in the presence of n-nucleotide and  $D$  sequence. However, we have at least  $4^M$  *in silico* sequences with the same features in the absence of  $D$  gene. Table 4.6 shows the total number of *in silico* sequences with the same features for a given n-nucleotide with length of  $M$  in the presence and absence of  $D$  sequence.

We can benefit from the above feature to perform parallel comparisons to accelerate the process. However, the level of parallelization is dependent on the length of n-nucleotide and  $D$  gene in the recombination process. The parallelization level affects the resource utilization and execution time performance. Increasing the parallelization level results in increasing the resource utilization directly as we need to provide a 134-bit register for every *in silico* sequences in the comparison unit. The execution time reduces as we increase the parallelization level from one to two/four. However, increasing the paral-

Table 4.6: The total number of *in silico* sequences with the same features for a given n-nucleotide's length with and without the *D* gene.

$N_{Length}$	<i>Total number of sequences in the presence of D gene</i>	<i>Total number of sequences in the absence of D gene</i>
0	1	1
1	8	4
2	48	16
3	256	64
4	1280	256
5	6144	1024
6	28672	4096
7	131072	16384
8	589824	65536
9	2621440	262144
10	11534336	1048576

lelization level from four to eight increases the critical path delay due to higher resource utilization. Also, this solution is not applicable for all the cases as shown in the table 4.6. Therefore, we set the parallelization level to four to have a general solution for most of the cases and avoid increasing the resource utilization. Note that we can not do any parallelization in the absence of n-nucleotide (length of n-nucleotide equal to zero). Thus, we detect this case at the second step using the  $n_{length}$  flag to perform the comparison process without parallelization.

*Step 2:* In this step, we first read one *in silico* sequence from the buffer. If the  $n_{length}$  of the received sequence is set to one, then we read three more *in silico* sequences from the buffer and proceed to the third step. Otherwise, we directly continue on to the third step.

*Step 3:* At this step, there is no difference between having one or four *in silico* sequences since all of them have the same features. Therefore, we first extract the  $sequence_{length}$  and  $group_{id}$  of the *in silico* sequence. Then, we calculate the index of  $starting_{array}$  and  $ending_{array}$  based on  $group_{id}$  and  $sequence_{length}$  using (4.2). Accordingly, we can access to the starting address and ending address of *in vivo* memory using the calculated index. Finally, we

set the address of *in vivo* memory bank to the starting address and proceed to the next step.

$$Index_{array} = group_{id} \times 19 + \frac{sequence_{length}}{2} - 2 \quad (4.2)$$

*Step 4:* In this step, we first read the *in vivo* sequence from the *in vivo* memory bank. Then, we compare the *in vivo* sequence with *in silico* sequences simultaneously. If there is a match between the *in silico* and *in vivo* sequences, then we set the address of counter memory bank to the current address of *in vivo* and proceed to the next step. Otherwise, we compare the current address of *in vivo* memory bank with the ending address of *in vivo* memory bank that is determined in the previous step. If it is match, then we go to the first step. Otherwise, we increment the current address of *in vivo* memory bank by one, read the next *in vivo* sequence, and do comparison.

*Step 5:* In this step, we first read the *counter* from the counter memory bank. Then, we update the *counter* value with the  $D_{path}$  and write the updated *counter* into the counter memory bank. At this point, if we find a match for all of the *in silico* sequences, then we move to the first step. Otherwise we continue on to the fourth step to do the comparison between the current address of *in vivo* memory bank and the ending address.

#### 4.4.4 First In First Out Buffer (FIFO)

We utilize FIFO in the PUs for three reasons. First, we can eliminate the communication overhead between the combination and comparison units, since the combination unit continuously generates the *in silico* sequence and passes it to the FIFO without considering the state of comparison unit. Also, the comparison unit can perform comparison as long as the buffer is not empty without considering the state of combination unit. Second, we can perform parallel comparisons using FIFO. Third, we can overlap the task of combination and comparison units using the FIFO, which results in reducing the execution time. The challenging decision for the FIFO relates to its size. The size of FIFO directly affects the BRAM utilization. We use 75% of the available BRAM in the target FPGA without the FIFO. Since our aim is to remain below 85%

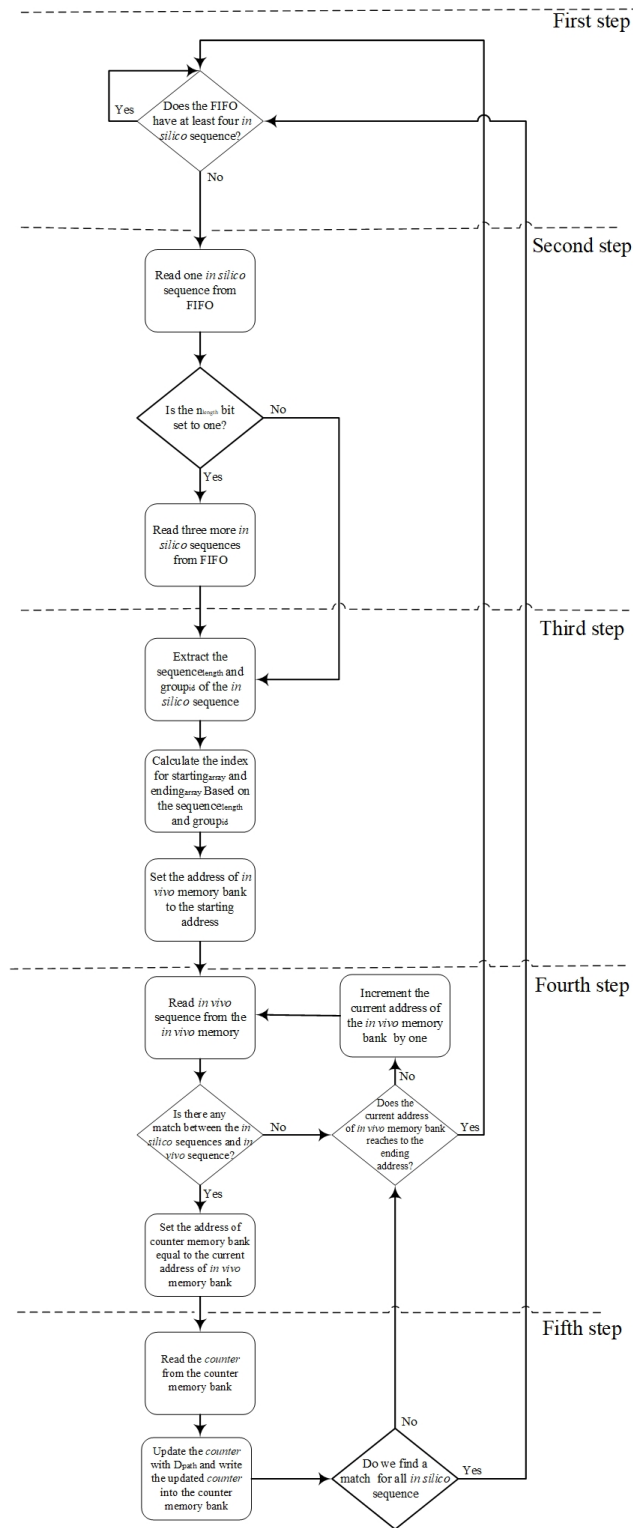


Figure 4.6: The algorithmic description for the structure of Comparison unit.

BRAM utilization, we can only assign  $1 \times 36Kb$  BRAM to each PU. Thus, we set the FIFO so that it can keep up to 256 *in silico* sequences with 134 bits.

#### 4.5 Simulation Results of the VJ Level Parallelization Architecture

Table 4.8 shows the resource utilization for the VJ level parallelization architecture. As shown, we utilize up to the 80% of the available resources on the target FPGA. The critical path delay is 9.328 *ns* and the maximum clock rate is 100 MHz. Table 4.4 shows the execution time for each  $N_{Length}$  for both n-level and VJ level parallelization approaches on Virtex-7 FPGA along with the execution time for the bit-wise GPU-based implementation on P100 GPU. The total execution time for the VJ level parallelization approach is equal to 8237 minutes ( $\sim 5.7$  days)) on the target FPGA. As stated in table 4.7, the total execution time for the VJ level approach reduces by a factor of 2.37 in comparison with the n-level parallelization approach. In addition, the execution time for n-nucleotide length of zero to five on FPGA is smaller than the bit-wise GPU-based implementation. However, for the large length of n-nucleotide, the execution time for GPU implementation is faster by a factor of up to 8. As a result the total execution time for the VJ level parallelization approach is slower than the baseline GPU-based implementation by a factor of 7.2.

#### 4.6 Drawback of VJ-Level Parallelization architecture

As shown in the previous sections, both FPGA implementations of the recombination process had a poor performance in comparison with the GPU implementation. The main difference between the GPU and FPGA implementation is the parallelization level for the comparison process. For the GPU-based implementation, all active threads compare their *in silico* sequences with one *in vivo* sequence. Therefore, the parallelization level at the comparison step is equal to the total number of active threads ( $4^m$ ). However, FPGA-based implementations did not offer this level of parallelization at the comparison stage. For the n-level approach on FPGA, we compare 256 *in silico* sequences



Table 4.7: Execution time for the n-level and VJ level parallelization approach for FPGA-based implementation using Virtex-7 in comparison with the bit-wise GPU-based implementations using P100.

$N_{Length}$	<i>Execution time (min) for n-level parallelization approach using Virtex-7</i>	<i>Execution time (min) for VJ level parallelization approach using Virtex-7</i>	<i>Execution time (min) for the bit-wise GPU-based implementation</i>
0	<1	<1	8.68
1	1	<1	9.34
2	1	<1	10.14
3	1	<1	10.92
4	1	<1	11.74
5	8	3	12.56
6	39	17	13.69
7	180	78	16.23
8	805	347	49.82
9	1768	1490	196.76
10	15243	6300	797.8
Total	19571	8237	1137.7

with one *in vivo* sequence and the parallelization level for comparison step is equal to 4 for the VJ-level approach. Therefore, we believe that the great performance of GPU implementation is due to the huge parallelization level for comparison process.

In order to evaluate our claim, we estimate the amount of time that is used for the combination and comparison process. For this purpose, we select one of the PC out of 120 units. In order to evaluate the worst case scenario, we chose the PC with the largest *in vivo* data set among all PC. Then, we calculate the total number of *in silico* sequences can be generated based on the selected PC (VJ pair). We also calculate the total number of required comparison for all possible *in silico* sequences in a given unit, assuming that there is no match between generated sequence and *in vivo* sequences. Simulation results show that, total of 3,517,387 *in silico* sequences can be generated for a selected PC and total number of comparison is equal to the 963,073,005 for that unit. As a result, the average of 271 comparisons are required for each *in silico* sequence in the worst case scenario. Assuming that combination unit generates one *in silico* sequence per clock cycle and comparison unit compares one *in vivo* sequence with four *in silico* sequences per clock cycle, then 98% of the total execution time is spend on the comparison process and only 2% of the exe-

Table 4.8: FPGA Resource Utilization for the VJ Level Parallelization Architecture.

<i>Component</i>	<i>Slice LUTs</i>	<i>Slice Registers</i>	<i>Slice</i>	<i>F7 Muxes</i>	<i>F8 Muxes</i>	<i>LUT Flip Flop Pairs</i>	<i>BRAM</i>
	<i>303600</i>	<i>607200</i>	<i>75900</i>	<i>151800</i>	<i>75900</i>	<i>303600</i>	<i>1030</i>
Combination unit	127080	22440	38400	0	0	13200	0
Comparison unit	93000	62520	30000	120	0	61440	0
FIFO	3840	4680	2040	0	0	2520	240
AGU	4440	2880	1560	0	0	2880	0
<i>V</i> memory bank unit	840	0	360	0	0	0	0
<i>J</i> memory bank unit	4800	0	3720	0	0	0	0
<i>D</i> memory bank unit	15000	0	4200	2880	1440	0	0
<i>in vivo</i> memory bank unit	0	0	0	0	0	0	452
<i>counter</i> memory bank unit	0	0	0	0	0	0	70
glue and display logics	1431	216	469	4	0	98	0
Total	245991	92736	68385	3004	2880	80040	762
Percent	81%	15%	90%	1.9%	3.7%	26%	73.9%

cution time is used for generating *in silico* sequences. Based on this analysis, the comparison process consumes the majority of the execution time and it is the main reason for the poor performance of FPGA-based implementation in comparison with the GPU-based implementation. To address this issue, we need to find a solution to reduce total number of required comparisons for each *in silico* sequence. One solution would be using hashing method for the comparison process. Hashing can reduce the lookup time from  $O(n)$  to  $O(k)$ , where  $n$  is the total number of *in vivo* sequences in a given data set, and  $k$  is the constant value significantly smaller than  $n$ .

## 4.7 Hashing Functions

In this section, we first study different hashing methods for the hardware implementation and select the one that is more suitable for our application. In 4.7.1, we provide a detailed explanation about the chosen hash function for the recombination process.

In [23], authors introduce shift-Add-XOR hash function which is based on bitwise AND, right and left shift, and addition. This hash function is suitable for hardware based implementation due to the nature of required operations for calculating the hash value. However, latency of this hash function depends on the input length which is not suitable for our application. Since, we are looking for a hash function which generate a hash value in one clock cycle. In [24], authors used CRC-style polynomials as a hashing function for NIDS pattern matching. The main advantages of CRCs based hashing function is that they involve simple functions (mainly XOR) with small implementation cost. However, the implementation cost for CRCs is heavily dependent on the input width. Another limitation of CRC based hashing is that a separate CRC generator is required for each specific sequence length which cause significant memory and logic cost. Also, the memory utilization is heavily dependent on the number of sequence of that length [24]. In [25], authors proposed variable length CRC hashing to eliminate the drawback of CRCs based hash function by using one CRC generator for the several close sequence lengths. However, this method is applicable to the data set that can guarantee the hash values

of first few characters of all sequence are unique. This is not applicable for our data set since the *in vivo* data set is portioned based on the VJ pair used to generate them as a result the first few character of all sequence is similar. In [26], authors utilize a hash function of class  $H_3$  with the hash table bins of CAM-based implementation [27] for virus checking application. The class of  $H_3$  only involves bit-wise AND and XOR operations which result in a fast hash function operation completing in one clock cycle. In addition, the class  $H_3$  hash function, has been proved to be very effective in distributing keys evenly among hash table entries [28] leading to a low collision rate. The logic cost for the hardware implementation of  $H_3$  hash function is low as it requires simple bitwise AND and XOR logics. We choose the hash function to be of class  $H_3$  due to its advantages over other hashing function.

#### 4.7.1 The class of function $H_3$

Let  $A = 0, 1, \dots, 2^n - 1$  be the key space and  $B = 0, 1, \dots, 2^m - 1$  be the address space, where  $n$  is the number of bits in the key and  $m$  is the number of bits in the address. Then, the class  $H_3$  is defined as follow: Let  $Q$  is the set of all  $n \times m$  matrices. For a give  $q \in Q$  and  $x \in A$ ,  $q(i)$  is the  $i$ th row of the matrix  $q$  and  $x(i)$  is the  $i$ th bit of  $x$ . The hashing function  $h_q(x) : A \implies B$  is defined as:  $h_q(x) = x_1.q(1) \oplus x_2.q(2) \oplus \dots \oplus x_i.q(i)$ , where  $.$  indicates bitwise AND operation and  $\oplus$  indicates bitwise exclusive OR operation.

Based on the definition of the class of  $H_3$ , we need  $n \times m$  AND modules and  $n$  XOR modules to map a  $n$ -bit key to a  $m$ -bit hash value on a target FPGA. Matrix  $q$  can also be stored in distributed ram for fast access.

### 4.8 Hardware Implementation of VJ-Level Parallelization Using Hash Function

Fig 4.7 shows the architecture of the proposed PC using the class of  $H_3$  hash function for the VJ level parallelization approach. In the following, we provide a detailed explanation of each unit in PE.

#### 4.8.1 Combination Unit

The inputs to the combination unit are the  $V$ ,  $J$  and  $D$  gene sequences received from their respective memory banks. The output of this unit are the 120-bits *in silico* sequence and the 5-bits  $D_{path}$  which are passed to the hash and comparison unit respectively. The structure of combination unit does not change significantly in comparison with previous architecture. The only change to this unit is that the length of the *in silico* sequence does not pass to the hash and comparison units. Since, comparison unit does not rely on the length information anymore.

#### 4.8.2 Hash Unit

The inputs to the hash unit is the 120-bit *in silico* sequence received from combination unit. The output is the 13-bit hash value which is used as an address for the indirect memory bank which we will explain later. Hash unit consists of  $120 \times 13$  AND modules and 120 XOR modules since the key space in our application is *in silico* sequences and their maximum length is 120 bits. Thirteen bits are required to address the *in vivo* memory bank unit since it contains at most 4762 *in vivo* sequences. Matrix  $q$  is randomly generated through a C code and hard coded to the distributed ram for fast access.

#### 4.8.3 Comparison Unit

The inputs for the comparison unit is 13-bit hash value received from the hash unit, 5-bit  $D_{path}$  received from combination unit indicating total number of path that *in silico* sequence can be generated, 120-bit *in silico* sequence received from combination unit, 128-bit *in vivo* sequence received from the *in vivo* memory bank, and the 16-bit counter indicating the number of times that specific *in vivo* sequence has been generated. The comparison unit consists of 120 bits comparator and indirect memory. Before we explain the structure of indirect memory, we need to explain the output of hash unit. Hash unit generate 13-bit hash value distributed between 0 to 8192. While, the *in vivo* memory bank at most contains 4762 sequences. If we use the result of hash

unit as an address to the *in vivo* memory bank unit, then we need to increase the size of memory which results in significant increase in memory utilization. Therefore, we decided to utilize indirect memory as a pointer to a possible matching sequence in the *in vivo* memory. Indeed, the indirect memory holds address for the *in vivo* memory bank unit corresponding to a given hash value. The memory utilization does not increase significantly since each entry is 13 bits in the indirect memory. Note that the indirect memory is initialized to -1 for those hash values that are not exist in the data set. This way, we can terminate comparison process quickly and without reading *in vivo* sequence.

As explained above, indirect memory generate an address associated with a given hash value for the *in vivo* memory bank unit. Then, the 120-bit comparator is used to compare the *in silico* with *in vivo* sequence. Finally, the counter memory is updated if there is a match, otherwise move on to the next *in silico* sequence.

#### 4.9 Simulation Results of the VJ Level Parallelization Architecture Using Hash Function

Table 4.10 shows the resource utilization for the VJ level parallelization architecture with hash unit. As shown, we utilize up to the 70% of the available resources on the target FPGA. Hash unit only utilize 4.5% of the available resources on target FPGA. In addition, resource utilization for combination and comparison units reduces by factors of  $1.09\times$  and  $7.9\times$  respectively in comparison with the VJ level architecture without hash function. This is due to the fact that we remove all logics corresponding with FIFO and starting and ending arrays. However, the memory utilization increase 4.9% in comparison with the VJ level architecture without hash function which is the results of employing indirect memory.

The critical path delay is 8.70 *ns* and the maximum clock rate is 100 MHz. Table 4.9 shows the execution time for each  $N_{Length}$  for VJ level parallelization with and without hash function on Virtex-7 FPGA along with the execution time for the bit-wise GPU-based implementation on P100 GPU. The total execution time for the VJ level parallelization approach using hash function is

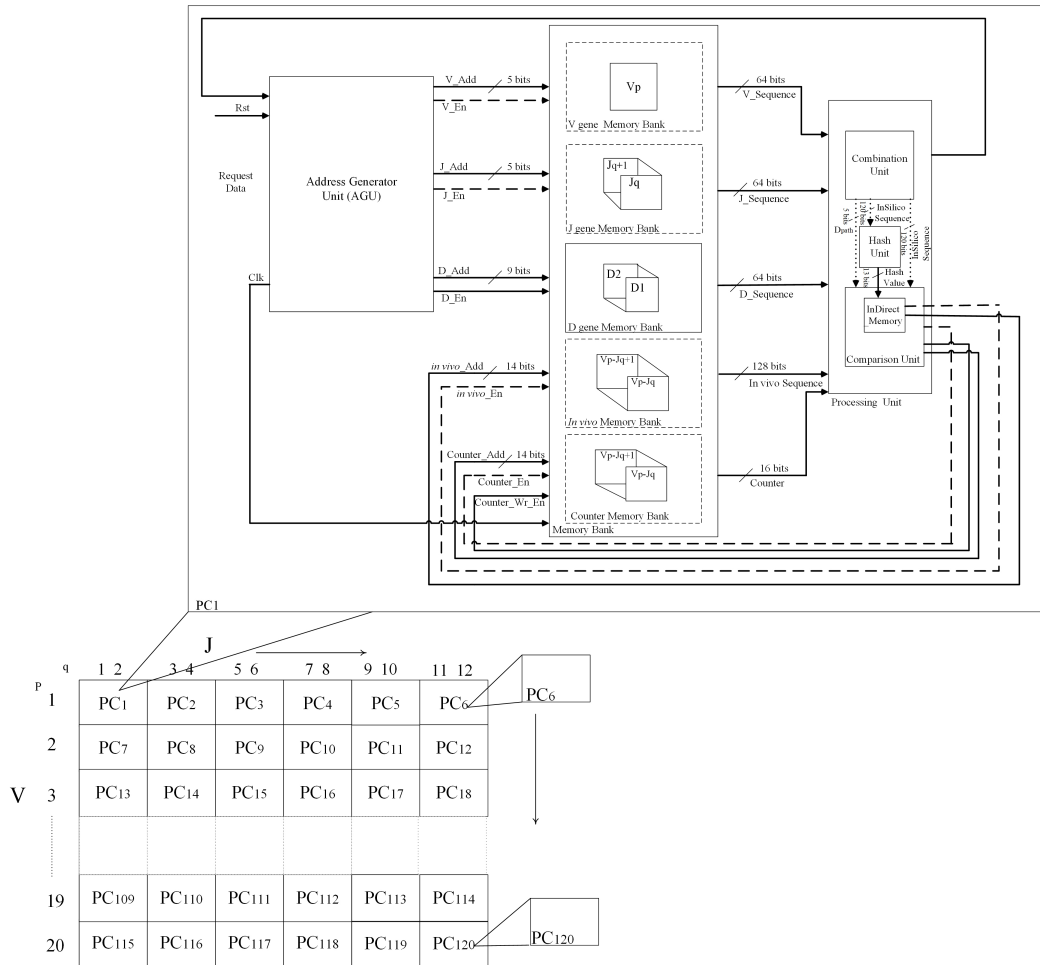


Figure 4.7: A high level view of hardware implementation of VJ level parallelization with hash unit for the  $V(D)J$  recombination process.

Table 4.9: Execution time for the VJ level parallelization approach with and without hash function for FPGA-based implementation using Virtex-7 in comparison with the bit-wise GPU-based implementations using P100.

$N_{Length}$	<i>Execution time (min) for VJ level parallelization approach without hash function</i>	<i>Execution time (min) for VJ level parallelization approach with hash function</i>	<i>Execution time (min) for the GPU-based implementation</i>
0	<1	<1	8.68
1	<1	<1	9.34
2	<1	<1	10.14
3	<1	<1	10.92
4	<1	<1	11.74
5	3	<1	12.56
6	17	2	13.69
7	78	8	16.23
8	347	38	49.82
9	1490	167	196.76
10	6300	726	797.8
Total	8237	947	1137.7

equal to 947 minutes ( $\sim 16$  hours) on the target FPGA. As stated in table 4.9, the total execution time for the VJ level approach using hash function reduces by a factor of  $8.69\times$  in comparison with the VJ-level parallelization approach without hashing. In addition, the VJ-level approach using hash function is  $1.2\times$  faster than the bit-wise GPU-based implementation.



Table 4.10: FPGA Resource Utilization for the VJ Level Parallelization Architecture Using Hash Function.

<i>Component</i>	<i>Slice LUTs</i> <b>303600</b>	<i>Slice Registers</i> <b>607200</b>	<i>Slice</i> <b>75900</b>	<i>F7 Muxes</i> <b>151800</b>	<i>F8 Muxes</i> <b>75900</b>	<i>LUT Flip Flop Pairs</i> <b>303600</b>	<i>BRAM</i> <b>1030</b>
Combination unit	116280	21840	33720	0	0	12240	0
Comparison unit	11760	4320	4080	0	0	3720	0
Hash Unit	13920	0	4200	0	0	0	0
AGU	4440	2880	1560	0	0	2880	0
<i>V</i> memory bank unit	840	0	360	0	0	0	0
<i>J</i> memory bank unit	4800	0	3720	0	0	0	0
<i>D</i> memory bank unit	15000	0	4200	2880	1440	0	0
<i>in vivo</i> memory bank unit	0	0	0	0	0	0	452
<i>counter</i> memory bank unit	0	0	0	0	0	0	70
<i>InDirect</i> memory	0	0	0	0	0	0	290
glue and display logics	1431	216	469	4	0	98	0
Total	168471	29256	52309	2884	1440	20378	812
Percent	55%	4.8%	68%	1.89%	1.89%	6.7%	78.8%

## CHAPTER 5

### Conclusion and Future Work

#### 5.1 Conclusions

The  $V(D)J$  recombination process in TCRs offers a diverse set of receptors, which is necessary to facilitate T-cell responses to foreign invaders. In addition, scrutiny of the TCR repertoire enable immunologists to understand the functionality of healthy immune system, determine the nature of successful and unsuccessful immune responses, and understand the immune mechanism in presence of different diseases. The response of immune system to specific antigen often leaves evidence in the form of repertoire sequence signatures that are common across individuals and these signature patterns can be associated with the corresponding antigen. Identification of these signatures help immunologists to understand the correlation between the immune receptors and different disease, which provides researchers the ability to identify immune receptor clones that can be converted into precision vaccines. However, analysis of TCR pool require modeling of the diverse set of TCR, which is computationally challenging as the total number of TCRs to be generated and processed can exceed  $10^{18}$  sequences. This massive scale of data processing poses as the barrier for immunologists to successfully understand the functionality of human immune system. Therefore, reducing the timescale of modeling the TCR repertoire is crucial for the immunologists.

In this dissertation, we introduced a bit-wise implementation of the  $V(D)J$  recombination algorithm, which reduces the constant memory and global memory footprint by factors of  $3.4\times$  and  $4\times$  respectively. On a single GPU, the bit-wise implementation reduces the total execution time by a factor of  $2.1\times$  compared to the baseline implementation. We presented the multi-GPU version of the bit-wise recombination and conducted availability analysis. We showed that beyond n-nucleotide length of eight, since we fully occupy the

thread blocks on a single GPU, we observed reduction in execution time with the increase in number of GPUs. However this reduction shows a saturating trend. We finally analyzed the root causes of observing a saturation trend in execution time reduction as we increase the GPU resources. As we transition from mouse data set to human data set, we expect the time scale of the experiments to increase by three orders of magnitude. In this scale, ability to reduce the simulation time from 40.5 hours to 18.9 hours on a single GPU and to 4.3 hours on a 8-GPU system for mouse data set is a significant gain that will allow us to count the number of unique pathways a TCR sequence can be generated, and conduct statistical analysis to correlate those frequently generated TCR sequences to certain diseases much faster than the baseline version.

We also mapped the  $V(D)J$  recombination algorithm onto FPGA and take advantage of the fine grained parallelism offered by the target FPGA whose architecture naturally matches the program architecture of the recombination process. We first map the recombination algorithm using the N-level parallelization approach, which is used for GPU-based implementation. We show that this implementation suffers a large critical path delay and as a result low clock frequency. In order to address the drawback of N-level architecture, we propose the VJ-level parallelization approach. Simulation results showed that the total execution time reduces by a factor of  $2.34\times$  in comparison with the N-level architecture for the FPGA-based implementation.

## 5.2 Future Work

Future work for the GPU-based implementation includes extending a bit-wise implementation of the  $V(D)J$  recombination process to explore a data containing 50 million sequences from 100 humans, as well as additional data sets from mice and other species. One of the potential future work involves proposing scalable implementation of recombination process for the multi-GPU environment. The scalable GPU-based implementation will enable immunologists to analyze the human data set and provide them with a more solid understanding of the mechanisms that control the recombination process in the human immune system.

For the FPGA-based implementation, one potential future work would be utilizing dictionary based algorithm to speed up the comparison process, which can result in accelerating the entire process. Another work would be employing *hash function* to eliminate the comparison step. This will significantly help us to increase the degree of parallelism due to elimination of comparators and significantly accelerate the process.

## REFERENCES

- [1] M. Gellert, "V(d)j recombination: Rag proteins, repair factors, and regulation," *Annual Review of Biochemistry*, vol. 71, no. 4, pp. 101–132, April 2002.
- [2] M. M. Davis, J. J. Boniface, Z. Reich, D. Lyons, J. Hampl, B. Arden, and Y.-h. Chien, "Ligand recognition by  $\alpha\beta$  t cell receptors," *Annual Review of Immunology*, vol. 16, no. 1, pp. 523–544, 1998.
- [3] Y. Ping, C. Liu, and Y. Zhang, "T-cell receptor-engineered t-cells for cancer treatment: current status and future direction," *Journal of Protein and Cell*, vol. 9, pp. 254–266, 2018.
- [4] B. Vincent, A. Buntzman, B. Hopson, C. McEwen, L. Cowell, A. Akoglu, H. Zhang, and J. Frelinger, "iwas-a novel approach to analyzing next generation sequence data for immunology," *Journal of Cellular Immunology*, vol. 299, pp. 6–13, January 2016.
- [5] R. Welsh, J. Che, M. Brehm, and L. Selin, "Heterologous immunity between viruses," *Journal of Immunological Reviews*, pp. 244–66, 2010.
- [6] G. Du, C. Chen, Y. Shen, L. Qiu, D. Huang, R. Wang, and Z. Chen, "Tcr repertoire, clonal dominance, and pulmonary trafficking of mycobacterium-specific cd4+ and cd8+ t effector cells in immunity against tuberculosis," *Journal of Immunology*, pp. 3940–3947, 2010.
- [7] D. Schatz and Y. Ji, "Recombination centers and the orchestration of v(d)j recombination," *Nature Reviews Immunology*, vol. 11, no. 4, pp. 251–263, 2011.
- [8] J. Mansilla-Soto and P. Cortes, "V(d)j recombination: artemis and its in vivo role in hairpin opening," *Journal of Experimental Medicine*, vol. 197, no. 5, pp. 543–547, 2003.
- [9] H. Li, C. Ye, G. Ji, and J. Han, "Determinants of public t cell responses," *Cell Research*, vol. 22, no. 1, pp. 33–42, 2012.
- [10] H. Robins, S. Srivastava, P. Campregher, C. Turtle, J. Andriesen, S. Riddell, C. Carlson, and E. Warren, "Overlap and effective size of the human cd8+ t cell receptor repertoire," *Sci Transl Med*, vol. 22, no. 47, pp. 47–64, 2010.
- [11] M. Basu, M. Hedge, and M. Mo, "Synthesis of compositionally unique dna by terminal deoxynucleotidyl transferase," *Biochemical and Biophysical Research Communications*, vol. 111, no. 3, pp. 1105–1112, 1983.

- [12] G. Gauss and M. Lieber, “Mechanistic constraints on diversity in human v(d)j recombination,” *Molecular and Cellular Biology*, vol. 16, no. 1, pp. 258–269, 1996.
- [13] V. Venturi, D. Price, D. Douek, and M. Davenport, “The molecular basis for public t-cell responses?” *Nature Reviews Immunology*, vol. 8, pp. 231–238, 2008.
- [14] A. Casrouge, E. Beaudoin, S. Dalle, C. Pannetier, J. Kanellopoulos, and P. Kourilsky, “Size estimate of the alpha beta tcr repertoire of naive mouse splenocytes,” *Journal of Immunology*, vol. 164, pp. 5782–5787, 2000.
- [15] V. Venturi, K. Kedzierska, D. Price, P. Doherty, D. Douek, S. Turner, and M. Davenport, “Sharing of t cell receptors in antigen specific responses is driven by convergent recombination,” *Proceeding National Academy of Sciences*, vol. 103, no. 49, pp. 18 691–18 696, 2006.
- [16] M. Quigley, H. Greenaway, V. Venturi, R. Lindsay, K. Quinn, R. Seder, D. Douek, M. Davenport, and D. Price, “Convergent recombination shapes the clonotypic landscape of the nave t-cell repertoire,” *Proceeding of the National Academy of Sciences*, vol. 107, no. 45, pp. 19 414–19 419, 2010.
- [17] G. Striemer, H. Krovi, A. Akoglu, B. Vincent, B. Hopson, J. Frelinger, and A. Buntzman, “Overcoming the limitations posed by tcr $\beta$  repertoire modeling through a gpu-based in-silico dna recombination algorithm,” in *IEEE Proceeding 28th Int. Parallel and Distributed Processing Symp*, pp. 231–240, May 2014.
- [18] NVIDIA, “Nvidia cuda c programming guide,” 2018. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- [19] A. Zapata and C. Amemiya, “Phylogeny of lower vertebrates and their immunological structures,” *Current Topics in Microbiology and Immunology*, vol. 248, pp. 67–107, 2000.
- [20] M. Lieber, “Site-specific recombination in the immune system,” *Journal of the Federation of American Societies for Experimental Biology*, vol. 5, pp. 2934–2944, 1991.
- [21] M. P. I. Forum, “Mpi: a message-passing interface standard version 3.0,” 2012.
- [22] NVIDIA, “Tesla p100 pcie gpu accelerator,” 2016. [Online]. Available: <http://images.nvidia.com/content/pdf/tesla/NV-tesla-p100-pcie-PB-08248-001-v01.pdf/>

- [23] M. Ramakrishna and J.Zobel, “Performance in practice of string hashing functions,” in *Proceeding international conference on Database Systems for Advanced Applications*, pp. 215–223, 1997.
- [24] G. Papadopoulos and D. Pnevmatikatos, “Hashing + memory = low cost, exact pattern matching,” *International conference on Field Programmable Logic and Applications*, pp. 39–44, 2005.
- [25] D. Pnevmatikatos and A. Arelakis, “Variable-length hashing for exact pattern matching,” *International conference on Field Programmable Logic and Applications*, pp. 1–6, 2006.
- [26] A. Fairouz and S. Khatri, “An FPGA-based coprocessor for hash unit acceleration,” *International Conference on Computer Design*, pp. 301–304, Nov 2017.
- [27] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (cam) circuits and architectures: A tutorial and survey,” *Journal of Solid-State Circuits*, vol. 41, no. 3, p. 712727, March 2006.
- [28] M. Ramakrishna, E. Fu, and E. Bahcekapil, “Efficient hardware hashing functions for high performance computers,” *Transactions on Computers*, vol. 46, no. 12, p. 1378 1381, Dec 1997.