Bit-wise and Multi-GPU Implementations of the DNA Recombination Algorithm

Elnaz Tavakoli Yazdi, Ankur Limaye, Ali Akoglu, Tosiron Adegbija and Adam Buntzman
Department of Electrical and Computer Engineering University of Arizona, Tucson, AZ 85721
Emails: {tavakoliyazdi, ankurlimaye, akoglu, tosiron, buntzman}@email.arizona.edu

***Abstract*—The $\mathbf{V(D)J}$ recombination 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. However, modeling TCR repertoire is computationally challenging as the total number of TCRs to be generated and processed can exceed $\mathbf{10^{18}}$ sequences. We propose a *bit-wise* implementation of the $\mathbf{V(D)J}$ recombination algorithm, which reduces the memory footprint and execution time by factors of 4 and 2, respectively, compared to the state-of-the-art GPU implementation. We also present a multi-GPU implementation, experimentally identify suitable workload partitioning strategies for both single- and multi-GPU implementations, and finally, expose the relationship between the 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 40.5 hours to 19 hours on a single GPU and 4.4 hours on an eight-GPU configuration.**

***Index Terms*—DNA recombination process, Graphics Processing Unit (GPU), bit-wise implementation, and multi-GPU.**

I. Introduction

The 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 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 to eliminate the 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 a healthy immune system, determining the nature of successful and unsuccessful immune responses, and understanding the immune mechanism in the presence of different diseases such as type 1 diabetes, various cancers (blood, breast, colorectal, etc.), rheumatoid arthritis (an autoimmune disease), and multiple sclerosis [3]. The response of the immune system to a specific antigen often leaves evidence in the form of repertoire sequence patterns (signatures) that are common across individuals. Ability to detect such patterns is critical for understanding the correlation between the immune receptors and different diseases, and identifying 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 detect a wide variety of antigens successfully. The immune systems of the vertebrates achieve this diversity through the DNA recombination process, known as the $V(D)J$ recombination [7]. This process involves a rearrangement of the 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 vary across different species, and it is more complicated in the humans than most vertebrates. For example, there are 20 different V genes in the mice, while there are 50 different V genes in humans. The combinations of V, D, and J gene segments of mice generate the TCR repertoire consisting of more than 10^{15} sequences. Furthermore, the total number of paths exhausted to generate all the combinations can exceed 10^{18} paths. Replicating the recombination process in a simulation environment allows immunologists to test different hypothesis on immune system response analysis. However, this simulation requires a massive scale of data processing.

The study by Striemer et al. [9], which successfully models the mouse TCRβ repertoire for the first time, shows that the time scale of the TCR synthesis can be reduced to 16 days on a single NVIDIA GTX480 GPU from an estimated execution time of 52 weeks on a general-purpose processor. To the best of our knowledge, this is the only work available for parallelizing the DNA recombination algorithm. We refer to this study as the *baseline implementation* for the remainder of this paper. The primary goal of modeling the TCR repertoire is to count the number of unique pathways that each *in vivo* sequence can be generated artificially by rearranging the input data set (V, D, J genes and n-nucleotide sequence). We aim to investigate ways to reduce the execution time and memory footprint of the recombination process using the mouse data set as a reference, so that we can establish a basis for rapidly modeling more complex systems. Towards this goal, compared to the state-of-the-art GPU-based implementation [9], we make the following contributions:

- Bit-wise implementation of the recombination process, consisting of fine-grained shift, concatenation, comparison, and counting over the binary domain input data set.
- Multi-GPU implementation with an even workload distribution across the GPUs.

The bit-wise implementation reduces the global and constant memory footprints by factors of 4 and 3.5, respectively, compared to the *baseline implementation* [9]. We show that the execution time of the *baseline implementation* reduces by a factor of 2.1 with the bit-wise implementation on a single NVIDIA Tesla P100 GPU. For the multi-GPU implementation,

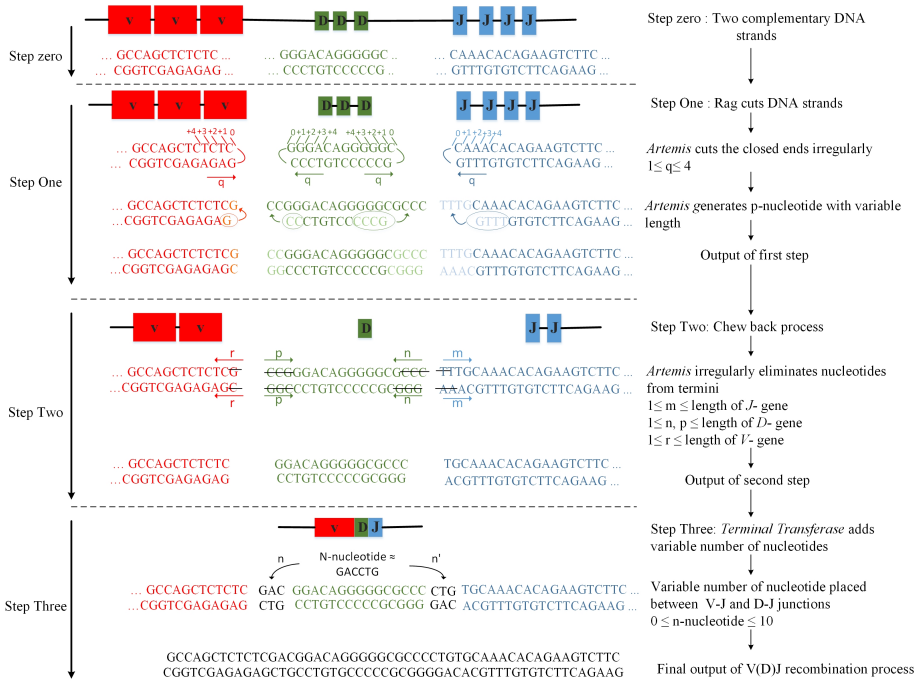

Fig. 1. $V(D)J$ recombination process showing the P-nucleotide formation with lengths: one for V–gene termini, two for left of D–gene termini, four for the right of D–gene termini, and four for J–gene termini in step one. The example depicts the elimination of one nucleotide on the V–gene termini, three nucleotides on both sides of the D–gene and two nucleotides on J–gene termini.

we introduce a *task generation* function that generates a unique task for each thread and eliminates the communication between the GPU threads. We conduct a scalability analysis and show that the execution time of the *baseline implementation* reduces by a factor of 9.2 with the multi-GPU and bit-wise implementation for the eight-GPU configuration.

The main hindrance to systematically developing new immunotherapies, new immunodiagnostics, and novel immunobiomarkers is the enormous magnitude ($> 10^{18}$ unique species) of the repertoire of TCR species that can be made by the immune system. Our bit-wise and multi-GPU implementation is an important step towards a TCR *in silico* synthesis model that is a practical option for selecting new cellular immunotherapies. The proposed GPU emulation of immune repertoire modeling brings this goal within grasp as we reduce the time scale of the simulations from days to hours.

The rest of the paper is organized as follows: In section II, we describe the DNA recombination algorithm from both the biological and algorithmic perspectives and explain the structure of the input data set. In section III, we discuss the related work for the recombination process. We explain the parallelization approach that is suitable for the GPU-based implementation in section IV. We present detailed explanations regarding the bit-wise representation and multi-GPU optimization strategies in sections V and VI, respectively. In section VII, we present our experimental environment followed by our evaluation strategy and simulation results in section VIII. Finally, we present the conclusion and future work in section IX.

II. DNA Recombination Algorithm

The $V(D)J$ recombination, as illustrated in Fig. 1, is a specialized DNA rearrangement process 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.

A. Biological Perspective

The TCRs are created by recombination of the V, D, and J gene segments. Fig. 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, a diverse set of sequences are generated. We summarize the three critical steps that contribute to this diversity in the following paragraphs.

In the first step, the recombination activation gene, *recombinase*, cuts the DNA at the joints between the V–D segment pairs and the D–J segment pairs. Immediately, the *template strand* and the *coding strand* bind to each other at the cut location. Subsequently, the *Artemis exonuclease* enzyme irregularly releases circular ends to generate a palindromic nucleotide (P-nucleotide) of variable lengths [7], [12], [15]. As shown in Fig. 1, at the beginning of Step 1, up to four genes from the *coding strand* are appended to the *template strand* on the right termini of the V segment, both termini of

the D segment, and the left termini of the J segment. This p-nucleotide addition of length up to four is one of the significant contributors to the diversity during the recombination.

In the second step, both strands of the V, D, and J segments go through a process called *chew back*. The *Artemis exonuclease* enzyme is involved in this *chew back* process too, in which a variable number of nucleotides are eliminated from the V, D, and J termini. As shown in Fig. 1, the *chew back* happens from right to left on the V segment, left to right on the J segment, and from both directions on the D segment. The *chew back* length ranges from one nucleotide to the length of that gene segment. This *chew back* process is the second contributor to the diversity.

In the third step, the *Terminal Transferase (TDT)* enzyme catalyzes the addition of n-nucleotides between the 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 the *in vivo* data set [9]. 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 the D and J termini to form D–J junction. Compared to the first two factors contributing to the diversity, having additional n-nucleotides between the V–D and D–J junctions grows the combinational search space enormously.

B. Algorithmic Perspective

We refer to the $V(D)J$ recombination as the $VnDn'J$ recombination, where V, D, and J indicate the unique sequences from each set of corresponding segments and n indicates the set of all possible nucleotide combinations. We refer to 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.

The four nucleotide bases A, G, C, and T, are used to generate an n-nucleotide sequence. Thus, for a nucleotide sequence of length m, there are 4^m unique nucleotide combinations. In the recombination process, these nucleotide sequences can be attached on either side or on both sides of the D sequence. To differentiate between the positions, we define the nucleotides as n and n'. The D sequence can cut the n-nucleotide at any position. Therefore, this complex junction-level combination may lead to the generation of an identical sequence via numerous ways. Our primary 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 six nested for loops. The first four loops iterate through all V, D, J and n sequences to form the *in silico* sequence. The fifth loop iterates through all possible combinations of nDn' sequences since the D sequence can cut the n-nucleotide at any position. All single sequences are combined and stored in the variable *Combination* through these nested loops. The last *for loop* iterates through all *in vivo* sequences and compares them against the *in silico* sequence. 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 has been exhausted.

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         for p =n-nucleotide_length to 0 do
            nDn' =
             CombineString(n[m][p], D[k], n[m][p−
             n-nucleotide_length])
            Combination =
             CombineString(V[i], nDn', J[j]);
6           for n = 0 to number of in vivo
              sequences do
                if Combination == invivo[n]
                 then
                  Counter[n]= Counter[n]+1;
                end
              end
            end
          end
        end
      end
    end
  end
```

Algorithm 1: V(D)J Recombination Algorithm

C. Input data sets

The input data sets consist of the 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 chew back and palindromic forms for each of the functional V, D, and J gene sequences need to participate in the recombination process for modeling the TCR repertoire, as illustrated in Fig. 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 process would generate 18 different sequences based on the *chew back* process (step two). The D and J gene data sets go through a similar process, as explained in section II-A; 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). The *in vivo* data set of C57BL/6 mice involves $101,822$ sequences, which are grouped based on the specific V–J pair used to generate that sequence. There is no other recombination path for an *in vivo* sequence other than the specific V–J pair that generates that specific sequence. We exploit this key feature to reduce the search space within the *in vivo* data set and hence reduce the execution time.

III. Related Work

The massive scale of data processing in TCR synthesis poses as a barrier for immunologists, which has led them to use computationally tractable statistical methods with a trade-off in accuracy. Earlier works on modeling the TCRβ repertoire in mice [16]–[19], commonly utilized a method that randomly sampled a subset of the TCR pool at a scale of 10^6 among 10^{15} recombinants, which has been shown to bias the simulations [21]. Our ability to evaluate all possible pathways allows us to investigate TCRs that would have the highest probability of participating in immune responses. The basis of our GPU implementation is the Convergent Recombination Hypothesis [20], and its functionality is verified with a one to one output match to the sequential algorithm [21].

The study by Striemer et al. [9] is the first work, which successfully modeled the entire TCRβ repertoire for the mouse dataset using a single GTX480 NVIDIA GPU. Their parallelization strategy is based on n-nucleotide level, where each thread is assigned a unique n-nucleotide sequence to apply the recombination process on that unique sequence. This approach ensured even workload distribution among the threads due to two reasons: First, each thread operated on a unique n-nucleotide to generate all possible *in silico* sequences using the same V, D, and J genes. Hence, the total number of recombination pathways were equal among all active threads. Second, since each thread performed the recombination process on the same V and J genes, the total number of *in vivo* sequences for the comparison process was the same for all active threads.

IV. Parallelization Strategy

In this section, we study the alternative parallelization approaches for the GPU-based implementation of the $V(D)J$ recombination process. We evaluate each strategy by answering the following questions: 1) What is the suitable workload distribution for 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, an alternative parallelization approach would be the V–J level parallelism, i.e., 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, as specified in Section II-A) along with all possible combinations of four bases for any given n-nucleotide length. However, since there are 20 V genes and 12 J genes, this implementation would require only 240 threads and result in significantly low thread utilization on the GPU. A finer granularity of V–J level parallelism can be realized by assigning one form (refer to the chew back 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, $102,446$ threads are required as there are 362 V genes and 283 J genes in the chew back and palindromic forms for the mice data set. This finer parallelization granularity occupies 90% of the threads on the target P100 series GPU.

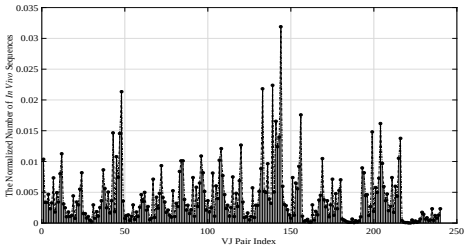

Fig. 2. Normalized distribution of *in vivo* sequences across 240 V–J pairs.

To evaluate the workload distribution of the fine-grained V–J level parallelization approach, we need to consider the workload for both *combination* and *comparison* steps. We refer to the *combination* step as the process of generating all possible *in silico* sequences for a given input data, and the *comparison* step as the process of comparing the generated sequences with *in vivo* sequences. The workload distribution for the *combination* step is even since each thread is assigned one form of V gene, one form of J gene, and both D genes. To evaluate the workload distribution of the *comparison* step, we provide a normalized distribution of *in vivo* sequences across 240 V–J pairs in Fig. 2. As shown, the total number of *in vivo* sequences is not evenly distributed across different V–J pairs, which directly affects the workload of each thread. Since, each thread needs to compare the generated *in silico* sequences against every *in vivo* sequence in the corresponding V–J pair, the V–J based assignment results in an uneven workload distribution among the GPU threads.

We evaluate the workload per thread for the *combination* steps of both approaches (n-nucleotide [9] and V–J methods) to decide which parallelization approach performs better in terms of the execution time. In the V–J level parallelism, each thread generates *in silico* sequences for all possible forms of n-nucleotide for lengths of zero to ten. A single thread needs to generate $706,042,015$ *in silico* sequences e process as there are $1,381,717$ unique n-nucleotide and 505 D gene sequences. 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, the workload per thread in V–J level parallelism is almost $13\times$ higher than the n-nucleotide level parallelization approach. Moreover, the n-nucleotide level parallelism offers a higher degree of data-level concurrency and allows representing the TCR synthesis process with up to 4^{10} independent threads for the n-nucleotide length of ten, whereas V–J level parallelism allows launching only $102,446$ threads. Consequently, the n-nucleotide-based approach offers better parallelization opportunity for the *comparison* process. Indeed, all active threads can compare their *in silico* sequences with the fetched *in vivo* sequence, which contributes to reducing the global memory access to one among all active threads for a specific *in vivo* sequence. We present our approach to bit-wise and multi-GPU implementations based on the existent n-nucleotide level parallelism in the following sections.

V. Bit-Wise Representation

A. Input data set conversion

The main objective of using bit-wise representation for mapping the recombination process is to reduce the memory footprint and execution time. We represent each base with two bits (A=00, C=01, T=10, G=11) and pack a sequence of four bases into a single byte. For a sequence (V, D, and J) whose length is not divisible by four, we pad zeros to the end of the sequence to make the length of the binary string a multiple of eight (one byte). For example, consider a V sequence having a length of ten characters (20 bits). We append four zeros to the end of the V sequence. As a result, the new V sequence has 24 bits and requires three bytes to store the data in the memory. We refer to the last byte of this V sequence, which has four zero-padded bits as the *padded bits*. We refer to the first two bytes containing the original bits of the V sequence as *full bytes*. We also store the length of each gene sequence along with the actual gene to distinguish between the *padded zeros* and the base *A*. The length of the gene is encoded into the first five bits of each gene sequence.

The maximum length of *in vivo* sequences is 60 characters. In the *baseline implementation* [9], the *in vivo* sequences are padded with zeros to make the length of all sequences equal to 64 bytes. This guarantees that each sequence allocation matches the number of threads in two warps, ensuring the memory alignment to realize coalesced memory accesses. In the bit-wise representation, we follow the same encoding procedure with padding and represent each *in vivo* sequence with a fixed size of 16 bytes.

In the *baseline implementation* [9], 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. The bit-wise representation allows us to store all the input data sets in constant memory alone, without having to resort to the global memory.

B. 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 the 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 approach reduces the number of memory accesses (global or constant) for a specific gene sequence to one among all active threads.

As discussed in section IV, the *in vivo* sequences are partitioned into 240 groups based on the V and J genes used to generate those sequences. The *baseline implementation* uses this feature to pare down the comparison search space [9]. The *in silico* sequences are compared only with the corresponding portion of the *in vivo* data set instead of the entire data set. We also use this feature in our implementation. Therefore, our GPU kernel, as illustrated in Algorithm 2, starts its execution by using V and J gene indexes to determine how many and which *in vivo* sequences 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.

```
Input  : V, J, D and in vivo sequences
Output : Number of times each unique in vivo
         sequence is generated (Counter)
1 for i = 0 to number of in vivo sequences do
    if threadIdx.x < 16 then
      SharedInVivo[threadIdx.x] = GlobalInVivo
       [i * 16 + threadIdx.x]
    end
2   for j = 0 to number of V sequences do
      if V[j] == SharedInVivo[length_{V[j]}] then
3       for k = 0 to number of D sequences do
4         for p =n-nucleotide_{length} to 0 do
            nDn' = (n[p], D[k], n[p'])
            if (V[j], nDn') ==
             SharedInVivo[length_{V[j]+n}] then
5             for m = 0 to number of J
               sequences do
                if (V[j], nDn', J[m]) ==
                 SharedInVivo then
                  Counter[i]= Counter[i]+1;
                end
              end
            end
          end
        end
      end
    end
  end
```

Algorithm 2: Pseudo-code for GPU Kernel

As shown in Algorithm 3, the task generation for each thread involves two nested *for loops*. The first *for loop* iterates through each byte of an n-nucleotide, one byte at a time. An

```
Input  : threadId, blockId, and blockDim
Output : n-nucleotide sequence
base[4] = {00, 01, 10, 11}
G_index = threadIdx.x + blockIdx.x * blockDim.x
1 for i = 0 to 3 do
2   for j = 0 to 9 increment by 2 do
      temp =
       base [{G_index + G_index/4^{4*i+((j-2)/2)}}%4] <<
       (8 - j)
      n-nucleotide[i] |= temp
    end
  end
```

Algorithm 3: Pseudo-code for the *task generator* function that generates a unique n-nucleotide sequence for each thread based on its thread and block indexes.

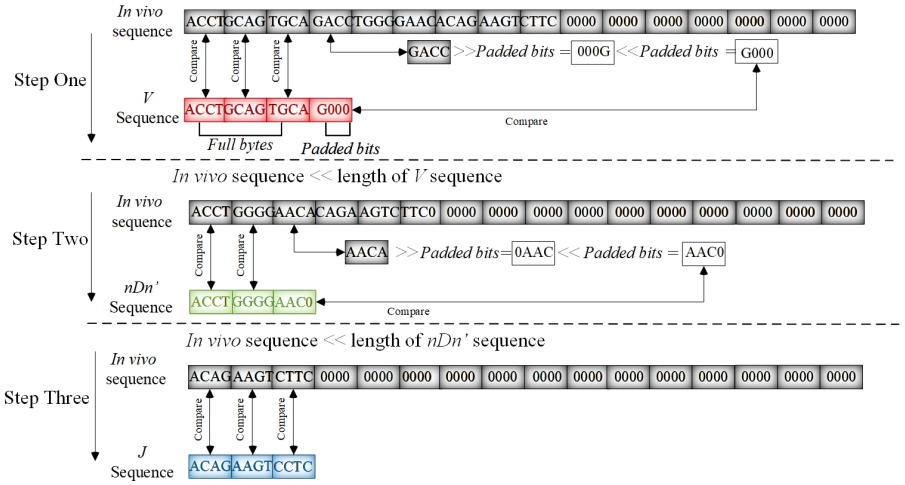

Fig. 3. Brief view of the comparison process for the bit-wise version of $V(D)J$ recombination process.

n-nucleotide is represented as a three-byte package since it can be up to 10 characters (20 bits) long. The second *for loop* iterates through the four bases in each byte of the n-nucleotide. When these two *for loops* are completely unrolled, a unique n-nucleotide of the given length *n* is assigned to each thread of the GPU. After assigning a unique task to each GPU thread, the recombination process begins on the GPU.

There are four main loops in the GPU kernel, as shown in Algorithm 2. 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 all the threads within a block share the *in vivo* sequence, we use *synchthreads()* to ensure 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 the threads within a block read the same V sequence from the constant memory while working 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 the global memory. Otherwise, we continue to compare the last byte of the V sequence with the pertinent byte of *in vivo* sequence. 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 refer to this process as an *alignment process*. Finally, we compare the last byte of the V sequence with the aligned byte of *in vivo* sequence. This procedure is shown in Step 1 of Fig. 3. 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 *for 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 section II. Therefore, each thread generates all possible combinations of nDn' sequence for a given D and n-nucleotide sequences. Then, each thread compares its 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 the V sequence. The comparison procedure is shown in Step 2 of Fig. 3, and it is same as the process explained for the V-loop. If there is a mismatch between the *in silico* and *in vivo* sequences, the thread terminates the current comparison, generates a new combination for the nDn' sequence and repeats the process. Otherwise, we continue to the next loop. It should be noted that, if a thread has generated all possible forms of the nDn' sequence for the given D and n-nucleotide sequences, we load a new D sequence and repeat the process.

The final *for 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, 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 3 of Fig. 3. 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 the 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, the 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.

VI. Multi-GPU Implementation

In n-nucleotide level parallelization, threads of a single GPU are assigned a unique n-nucleotide sequence while they work on the same V and J gene. From the multi-GPU implementation perspective, we define a global index for each thread based on its thread, block, and GPU indexes along with

TABLE I
P100 GPU STREAMING MULTIPROCESSOR FEATURES

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

the GPU dimension as shown in (1) to generate a unique n-nucleotide sequence for each active thread and utilize a *task generator* function that is presented in Algorithm 3.

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

$$G_{\text{index}} = \text{threadIdx} + (\text{blockIdx} \times \text{blockDim}) + (\text{GPUIdx} \times \text{GPUDim}) \quad (1)$$

In order to distribute the workload evenly among GPUs, we first calculate the total number of required threads, which is 4^m, where m is the length of the n-nucleotide sequence. Then, we calculate the total number of required blocks based on the thread-block configuration (refer to Fig. 4). Finally, we calculate the total number of blocks in each GPU using (2). The -1 in the nominator and +1 in the denominator are used for a configuration with an odd number of GPUs. For example, we need $262,144$ (4^9) threads for an n-nucleotide length of nine. As we present in section VIII, the configuration having 2048 blocks and 128 threads per block is the desired configuration on a single GPU. For a two-GPU implementation, based on (2), each GPU is assigned 1024 blocks with 128 threads in each block. This assignment ensures that the workload distribution among the GPUs and GPU threads are equal.

$$\#\text{blocks} = \frac{\#\text{total threads} - 1}{\#\text{threads per block} \times \#\text{GPUs} + 1} \quad (2)$$

We perform a reduction process (all-reduced) to obtain the final result from multiple GPUs. This process accumulates all the results at the root node and copies them to the global memory of the host. We note that the size of all-reduce depends on the total number of *in vivo* sequences for a given V–J pair. The largest size of V–J pair contains 3249 *in vivo* sequences; hence, we need to reduce 6 KB data at most.

VII. EXPERIMENTAL SETUP

We conducted our experiments on a cluster consisting of NVIDIA P100 GPU accelerator [14]. The system is composed of Intel Haswell V3 28 core processor nodes, featuring 192 GB RAM per node, in which 8 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 the node to storage interconnect. Table I 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. For the bit-wise implementation of the $V(D)J$ recombination algorithm with n-level granularity, each thread utilizes 48 registers, while there are $65,536$ registers available per SM. Therefore, the maximum number of active threads per SM is 1365 due to the register usage constraint.

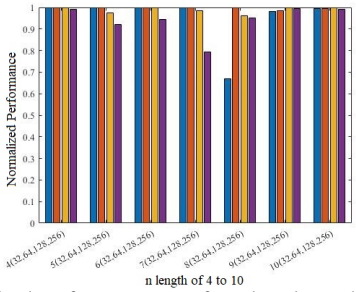

Fig. 4. Normalized performance over four threads per block configurations (32, 64, 128, 256) for each n-nucleotide length ranging from four to ten.

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 V, the shared memory usage per block is 16 bytes plus one byte per thread for the counter value storage. Thus, if we consider a block size of 128 threads, only 134 bytes of shared memory is required per thread block, allowing for 489 thread blocks per SM. Given that for the n-nucleotide length of nine with 2048 blocks and 128 threads per block configuration, there are only 10 active threads per SM due to the register usage constraint. As a result, we do not reach the limiting factor (489 thread blocks per SM) for the shared memory usage.

VIII. 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 times of our bit-wise implementation with the *baseline implementation* [9] on the same experimental setup for each n-nucleotide length. Finally, we present the execution time analysis for the multi-GPU implementation with up to eight nodes.

A. Thread Block Configuration Analysis

Fig. 4 shows the normalized execution time results for 32, 64, 128, and 256 threads per block configuration for each n-nucleotide length ranging from four to ten. For each length of the n-nucleotide sequence, we use the shortest execution time to normalize the execution times of other configurations. Therefore, a normalized value of 1 represents the best performance for a given length. We did not consider the n-nucleotide lengths of zero to three as the number of threads are not sufficient to utilize multiple *warps* executing concurrently. As shown in Fig. 4, the differences between the performance of various thread block configurations are negligible for the lengths of four to six since the workload amount not sufficient

TABLE II
THREAD AND WARP UTILIZATION FOR AN N-NUCLEOTIDE LENGTH OF SEVEN FOR THREE THREAD BLOCK CONFIGURATIONS. THE VALUES IN PARENTHESIS INDICATE THE PERCENTAGE OF UTILIZATION

Thread block configuration	Threads per SM	Thread blocks per SM	Warp
64	1344 (65.0%)	21 (65.00%)	42 (65.0%)
128	1280 (62.5%)	10 (31.25%)	40 (62.5%)
256	1280 (62.5%)	5 (15.62%)	40 (62.5%)

to utilize all the available multiprocessors of the P100 GPU. For the n-nucleotide lengths of less than seven, the thread utilization is below 14% as the total number of required threads is less than 2^{14}, while $114,688$ threads are available in the P100. However, for an n-nucleotide length of greater than seven, the workload increases such that more than 60% of the available GPU threads are utilized.

There is a 20% performance degradation for the n-nucleotide length of seven for the 256 threads per block configuration compared to other configurations. For the n-nucleotide length of seven, the recombination process completes in one iteration for all thread block configurations 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 II.

For the n-nucleotide length of eight, if 64 threads per block are employed, we can have maximum $1,365$ active threads per SM based on the register usage constraint, while based on the thread block configuration, we can have a maximum of 21 blocks with 64 threads. This results in a total of $75,264$ $(21 \times 64 \times 56)$ threads, which is greater than the required number of threads for an n-nucleotide length of eight. Therefore, the recombination process completes in one iteration for 64 threads per block configuration, while it can not be completed in one iteration with 32 threads per block. 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 completes in one iteration for all the three configurations.

For the 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 result in completing the recombination process in more than one iteration. For the 64, 128, and 256 threads per block configurations, four iterations are 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. For the n-nucleotide length of ten, for all threads per block configurations, the GPU is fully utilized, and execution takes the same number of iterations. Thus, we observe a negligible difference in terms of execution time among them.

In summary, based on Fig. 4, we set 64 threads per block configuration for the n-nucleotide lengths four to eight, and 128 for lengths nine and ten. In the following subsection, we evaluate the performance of bit-wise and multi-GPU implementations compared to the *baseline implementation*.

TABLE III
THE MEMORY FOOTPRINT FOR THE BIT-WISE IMPLEMENTATION IN COMPARISON WITH THE BASELINE APPROACH

Gene	Memory	Baseline [9] (byte)	Bit-wise (byte)	Reduction (%)
V	Constant	1448	425	70.65
J	Constant	3107	913	70.61
D	Constant	3210	908	71.71
in vivo	Global	6517568	1629392	75.00

TABLE IV
EXECUTION TIME (IN MINUTES) ON A SINGLE GPU: BASELINE VS. BIT-WISE IMPLEMENTATIONS

N-length	Baseline	Bit-wise
0	8.36	9.16
1	10.17	9.82
2	12.57	10.62
3	15.38	11.40
4	18.47	11.95
5	21.73	13.04
6	25.67	14.17
7	32.09	16.71
8	102.03	50.30
9	426.90	197.24
10	1755.35	798.20
Total	2428.72	1142.61

B. Bit-wise Simulation Results

In order to evaluate the bit-wise implementation, we first executed the *baseline implementation* on a single Tesla P100 GPU. We use the timing results and memory footprint of the *baseline implementation* as the reference points for performance comparison.

Table III shows the total amount of required memory for V, D, and J genes that are stored in the constant memory, and *in vivo* data set stored in the global memory by the *baseline* and bit-wise implementations. As stated in Table III, 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 IV shows the execution times for the n-nucleotide lengths from zero to ten for the *baseline* and bit-wise implementations. The last row shows that the total execution time for the recombination process is $2.1\times$ faster with the bit-wise implementation compared to the *baseline implementation*. Note that the execution time for the conversion process is 29 seconds. We include this one time cost across all n-nucleotide lengths in the total execution time. For the n-nucleotide length of eight, we utilize 87.5% of the available SMs on a single GPU with a thread block configuration of 64 (shortest execution time). After the n-nucleotide length of eight, the execution time increases by about a factor of 4 for each increment of n-nucleotide length since the SMs become fully utilized and the execution turns into an iterative flow.

C. Multi-GPU Simulation Results

Table V 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

TABLE V
EXECUTION TIME IN MINUTES FOR EACH N-NUCLEOTIDE LENGTH (0 TO 10) WITH RESPECT TO THE NUMBER OF GPUS (1 TO 8) FOR THE BIT-WISE IMPLEMENTATION

N-length	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs	7 GPUs	8 GPUs
0	9.16	9.71	9.76	9.57	9.57	9.57	9.58	9.57
1	9.82	10.39	10.45	10.25	10.25	10.26	10.26	10.26
2	10.62	11.17	11.23	11.03	11.04	11.04	11.03	11.04
3	11.40	11.96	12.03	11.82	11.83	11.83	11.83	11.83
4	11.95	12.77	12.85	12.60	12.61	12.60	12.61	12.60
5	13.04	13.66	13.72	13.51	13.51	13.49	13.49	13.45
6	14.17	14.82	14.87	14.39	14.38	14.38	14.37	14.36
7	16.71	17.09	16.14	15.91	15.80	15.76	15.74	15.71
8	50.30	28.37	23.25	19.40	18.10	18.05	18.03	18.00
9	197.24	113.34	82.48	59.08	47.07	40.70	34.84	29.18
10	798.20	456.21	301.66	231.85	185.79	159.96	140.10	116.70
Total	1142.61	699.49	508.44	409.41	349.95	317.64	291.88	262.70

trends in execution time improvement with respect to change in the number of GPUs.

The key observation from Table V is that the execution time increases slightly if multiple GPUs are utilized for n-nucleotide lengths that are less than 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 2048×56 $(57,344)$ active threads. Moreover, the extra reduction step during the multi-GPU execution introduces a slight execution time overhead. For example, we notice this overhead with the two-GPU implementation where execution time is higher compared to the single GPU implementation for the n-nucleotide lengths of up to seven. The reduction overhead is compensated for larger n-nucleotide lengths as we observe smaller execution times as we increase the number of GPUs.

We observe a reduction in the execution time with multiple GPUs for n-nucleotide lengths greater than seven. This happens because a single GPU is almost fully utilized at 87.5% for an n-nucleotide length of more than seven, as explained in section VIII-B. Since the required number of threads exceed the active thread count per GPU, we observe the benefit of the multi-GPU implementation for n-nucleotide lengths of eight or more. For these lengths, we expect to observe a relatively linear reduction in the execution time for a given n-nucleotide length as we increase the number GPUs. However, for the n-nucleotide length of eight, the simulation results show a saturating trend in execution time reduction where adding another GPU resource not reduce the execution time beyond four GPUs. For the 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 $(ceil(4^9/1365 \times 56))$ using a single GPU. In this case, there are $32,824$ active threads in the last iteration utilizing only 29% of the GPU threads. Employing two GPUs results in completing the process in two iterations, with $54,632$ threads in the last iteration. In this case, during the last iteration, we are only utilizing 47% of the threads on each GPU. Therefore, we do not observe a $2\times$ speed up with two GPUs. Utilizing four GPUs for the n-nucleotide length of nine results in completing the process in one iteration with the thread utilization for each GPU being 85.7%. Beyond this point, we observe a linear reduction in execution time with respect to an increase in the number of GPUs.

For the n-nucleotide length of ten, the required number of threads is 4^{10} $(1,048,576)$. The recombination process is completed in 14 iterations $(ceil(4^{10}/1365 \times 56))$ using a single GPU while there are $54,856$ active threads in the last iteration (47% GPU thread utilization). However, employing two GPUs results in completing the process in 7 iterations while the GPU thread utilization is at 57% for each GPU in the last iteration. Using three GPUs results in completing the recombination process in 5 iterations while the GPU thread utilization is at 38.2% in the last iteration. When we employ four GPUs, iteration count becomes 4 while each GPU is at 28% thread utilization in their last iteration. For the configurations with two to four GPUs, the underutilization of the threads during the last iteration is the root cause for not observing a linear reduction in the execution time with respect to the increase in GPU count. However, beyond the four GPUs configuration, we observe an almost linear reduction in execution time.

For the single GPU version, in the section VIII-B, we showed that execution time increased by about a factor of 4 at each increment 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 V, the execution time using two GPUs for the n-nucleotide length of nine is about $4\times$ the execution time for the n-nucleotide length of eight. We observe a factor of about 4 consistently as we increase the length from nine to ten for all GPU configurations. Furthermore, we should expect the similar execution times 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 V, the execution time for the n-nucleotide length of nine is 197 minutes for a single GPU. However, we observe that execution time as 231 minutes for n-nucleotide of ten with four GPUs. We identify three factors for this discrepancy. The first factor is the overhead of the reduction process with the increase in the number of GPUs that was explained earlier in this section. The second factor is the difference between

the total number of nDn' combinations, which indicates the number of different cuts that can be applied by a D sequence. As stated in Section II, an n-nucleotide sequence can be cut by a D sequence at any position, and each thread needs to generate all possible combinations of D with an n-nucleotide sequence. As the length of n-nucleotide increases by one, the total possible combinations of nDn' increases by one for the 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. The third factor is the variation in the number of times each thread finds a match in the database or terminates early. Similarly, 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 four GPUs. This discrepancy is about 2.6 minutes for the length pair of seven and eight with one and four GPUs respectively. We also attribute this discrepancy in reduction trend to the three factors listed above.

IX. Conclusions and Future Work

In this paper, we introduce 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 then present the multi-GPU version of the bit-wise implementation and conduct a scalability analysis. We show that beyond n-nucleotide lengths of eight, we observe a reduction in execution time with the increase in the number of GPUs since we fully occupy the thread blocks on a single GPU. 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, the ability to reduce the simulation time from 40.5 hours to 19 hours on a single GPU and to 4.4 hours on an eight-GPUs 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 should also mention that as we move from the mouse to human data set, the number of V, D, J genes along with the *in vivo* sequences increases. More importantly, the length of n-nucleotide is expected to grow up to 14 [4] to be able to model the entire TCR repertoire. Our bit-wise and multi-GPU implementations enabled with the n-level parallelization approach is still applicable to the human data set, with increased workloads per thread due to larger combinatorial search space for longer n-nucleotide lengths.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) research project under award number CNS-1624668.

References

[1] M. Gellert, " V(D)J Recombination: RAG Proteins, Repair Factors, and Regulation," *Annual Review of Biochemistry*, vol. 71, pp. 101–132, July 2002.

[2] M. M. Davis, J. J. Boniface, Z. Reich, D. Lyons, J. Hampl, B. Arden, and Y. Chien," Ligand recognition by $\alpha\beta$ T cell receptors," *Annual Review of Immunology*, vol. 16, pp. 523–544, April 1998.

[3] Y. Ping, C. Liu, and Y. Zhang," T–cell receptor–engineered T cells for cancer treatment: current status and future direction," *Protein and Cell Journal*, vol. 9, pp. 254–266, March 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," *Cellular Immunology*, vol. 299, pp. 6–13, January 2016.

[5] R. M. Welsh, J. W. Che, M. A. Brehm, and L. K. Selin, "Heterologous immunity between viruses," *in Immunological Reviews Journal*, pp.244–266, April 2010.

[6] G. Du, C. Y. Chen, Y. Shen, L. Qiu, D. Huang, R. Wang, and Z. W. Chen, "TCR repertoire, clonal dominance, and pulmonary trafficking of mycobacterium-specific CD4+ and CD8+ T effector cells in immunity against tuberculosis," *Journal of Immunology*, vol. 185, pp. 3940–3947, October 2010.

[7] D. G. Schatz, and Y. Ji, "Recombination centers and the orchestration of V(D)J recombination," *Nature Reviews Immunology*, vol. 11, No. 4, pp. 251–263, April 2011.

[8] J. MansillaSoto, and P. Cortes,"V(D)J recombination: artemis and its in vivo role in hairpin opening," *The Journal of Experimental Medicine*, vol. 197, no. 5, pp. 543–547, April 2003.

[9] G. Striemer, H. Krovi, A. Akoglu, B. Vincent, B. Hopson, J. Frelinger, and A. Buntzman, "Overcoming the limitations posed by TCRβ repertoire modeling through a GPU-Based in-silico DNA recombination algorithm," *in Parallel and Distributed Processing Symposium*, pp. 231–240, May 2014.

[10] NVIDIA, "NVIDIA CUDA C programming guide," 2018.

[11] M. M. Davis, and P. J. Bjorkman, "T-Cell Antigen Receptor Genes and T-Cell Recognition," *Nature*, vol. 334, pp. 395–402 October 1988.

[12] A. Zapata, and C. Amemiya, "Phylogeny of lower vertebrates and their immunological structures," *Current Topics in Microbiology and Immunology*, vol. 248, pp. 67–107, October 2000.

[13] M. Pedemonte, E. Alba, and F. Luna, "Bit-wise operations for GPU implementation of genetic algorithms," *The Proceeding on Genetic and evolutionary computation*, pp. 439–446, July 2011.

[14] NVIDIA, "TESLA P100 PCIe gpu accelerator," *http://images.nvidia.com/content/pdf/tesla/NV-tesla-p100-pcie-PB-08248-001-v01.pdf/*, 2016.

[15] L. MR, "Site-specific recombination in the immune system," *The Journal of the Federation of American Societies for Experimental Biology*, vol. 5, pp. 2934–2944, November 1991.

[16] V. Venturi, H. Chin, D. Price, D. Douek, and M. Davenport, "The role of production frequency in the sharing of simian immunodeficiency virus-specific CD8+ TCRs between macaques," *Journal of Immunology*, vol. 181, no. 4, pp. 2597–2609, May 2008.

[17] B. D. Rudd, V. Venturi, M. P. Davenport, and J. Nikolich-Zugich, "Evolution of the antigen-specific CD8+ TCR repertoire across the life span: evidence for clonal homogenization of the old TCR repertoire," *Journal of Immunology*, vol. 186, no.4 pp. 2056–2064, March 2011.

[18] M. V. Pogorelyy, A. A. Minervina, D. M. Chudakov, I. Z. Mamedov, Y. B. Lebedev, T. Mora, and A. M. Walczak, "Method for identification of condition-associated public antigen receptor sequences," *Journal of Elife*, vol. 7, no. 13, March 2018.

[19] E. Yuval, S. Zachary, C. Curtis, G. Mora, Thierry, and W. Aleksandra, "Predicting the spectrum of TCR repertoire sharing with a data-driven model of recombination," *Immunology Reviews*, vol. 284, no. 1, pp.167–179, July 2018.

[20] V. Venturi, K. Kedzierska, D. A. Price, P. C. Doherty, D. C. Douek, S. J. Turner, and M. P. Davenport, "Sharing of T Cell Receptors in Antigen Specific Responses is Driven by Convergent Recombination ," *National Academy of Sciences*, vol. 103, no. 49, pp. 18691–18696, April 2006.

[21] M. F. Quigley, H. Y. Greenaway, V. Venturi, R. Lindsay, K. M. Quinn, R. A. Seder, D. C. Douek, M. P. Davenport, and D. A. Price, "Convergent Recombination Shapes the Clonotypic Landscape of the Nave T-cell Repertoire," *National Academy of Sciences*, vol. 107, no. 45, pp. 19414–19419, April 2010.