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

Anonymous Author(s)

ABSTRACT

The $V(D)J$ recombination is the primary mechanism for generating a diverse repertoire of T-cell receptors (TCRs) essential to 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 10^{18} sequences. We propose a *bit-wise* implementation of the $V(D)J$ 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-based implementation. We present 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 *bit-wise* implementation reduces the execution time from 40.5 hours to 18.9 hours on a single GPU and to 4.3 hours on an 8-GPU configuration.

KEYWORDS

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

ACM Reference Format:

Anonymous Author(s). 2019. Bit-wise and Multi-GPU Implementations of the DNA Recombination Algorithm. In *Proceedings of ACM Conference (SC19).* SC19, Colorado, CO, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [3]. A TCR recognizes an antigen by detecting the small protein fragments that are on the surface of that antigen, and then sends a massage to the nucleus of its T-cell. This successful recognition induces a response for eliminating antigens [1]. 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 [8]. 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 [2, 12, 13].

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 [9]. 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 [4, 9]. 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. Combinations of V, D, and J gene segments of mice generates the TCR repertoire consisting of more than 10^{15} sequences. Furthermore, the total number of paths exhausted to generate each combination can exceed 10^{18}. Replicating the recombination process in simulation environment allows immunologists to test different hypothesis on immune system response analysis. However this simulation requires massive scale of data processing.

The study by Striemer et al. [10], which successfully models the mouse β-TCR repertoire for the first time, shows that the time scale of the TCR synthesis can be reduced to 16 days on a single NVIDIA GTX 480 GPU from an estimated execution time of 52 weeks on a general purpose processor. We will refer to this study as the *baseline implementation* for the remainder of this paper. When transitioning from the mouse model to more clinically relevant datasets (e.g. human patients), the computational workload will increase as the TCR repertoire increases by three orders of magnitude to 10^{18} [12]. To be able to cope with simulations at this scale, 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:

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

CUDA natively supports 32-bit integer shift and 32-bit bitwise "AND", "OR", and "XOR" operations [7], which allows us to implement the bitwise version of recombination process as it heavily

SC19, November 2019, Denver

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

relies on shift, concatenation(using "shift" and "or") and comparison ("xor") operations. In order to utilize the bit-wise operations, we convert the input data set from character based representation of four types of bases in DNA (A, G, C, T) to binary based (2-bits per character) representation 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, as 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 lengths 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 the GPU to avoid using padded bits of the input data that are not aligned with the byte addressing. For the multi-GPU implementation, we first 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. We then define an *indexing* function, which generates global index for threads in different GPUs based on the index of GPU and the GPU dimension.

Overall, our aim is to count number of paths 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 that thread completes the task of generating all possible sequences based on its given n-nucleotide. The two-bit based representation reduces the global and constant memory footprints by factors of 4 and 3.5 compared to the *baseline implementation* [10, 11] respectively. We show that we reduce the execution time of the *baseline implementation* by a factor of 2.1 through bit-wise implementation on a single NVIDIA Tesla P100 GPU, and by a factor of 4.4 through multi-GPU implementation on the eight GPU configuration.

The pressing need for reducing the timescale of immunotherapy studies makes the 2.1x speedup of the GPU implementation a significant 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. The rest of the paper is organized as follows: In section 2, we describe the DNA recombination algorithm from both biological and algorithmic perspective, and explain the structure of input data set. We explain parallelization approach that is suitable for the GPU-based implementation in section 3. We provide a detailed explanation regarding the bit-wise representation and multi-GPU optimization strategies in sections 4 and 5 respectively. In section 6, we explain our experimental environment. We describe our evaluation strategy and simulation results in section 7. Finally, we present the conclusion and future work in section 8.

2 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.

2.1 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, 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 [5, 9, 14]. From the starting point of the arrow shown in step one of Fig. 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.

In the second step, both strands of the V, D and J segments go through a process called chew back. Once more, the *Artemis exonuclease* enzyme is involved in this chew back process, in which a variable number of nucleotide are eliminated from the V, D and J termini. The chew back 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 chew back 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 the *in vivo* data set [10]. 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.

2.2 Algorithmic Perspective

We refer to the $V(D)J$ recombination as $VnDn'J$ recombination from algorithmic discussions 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 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.

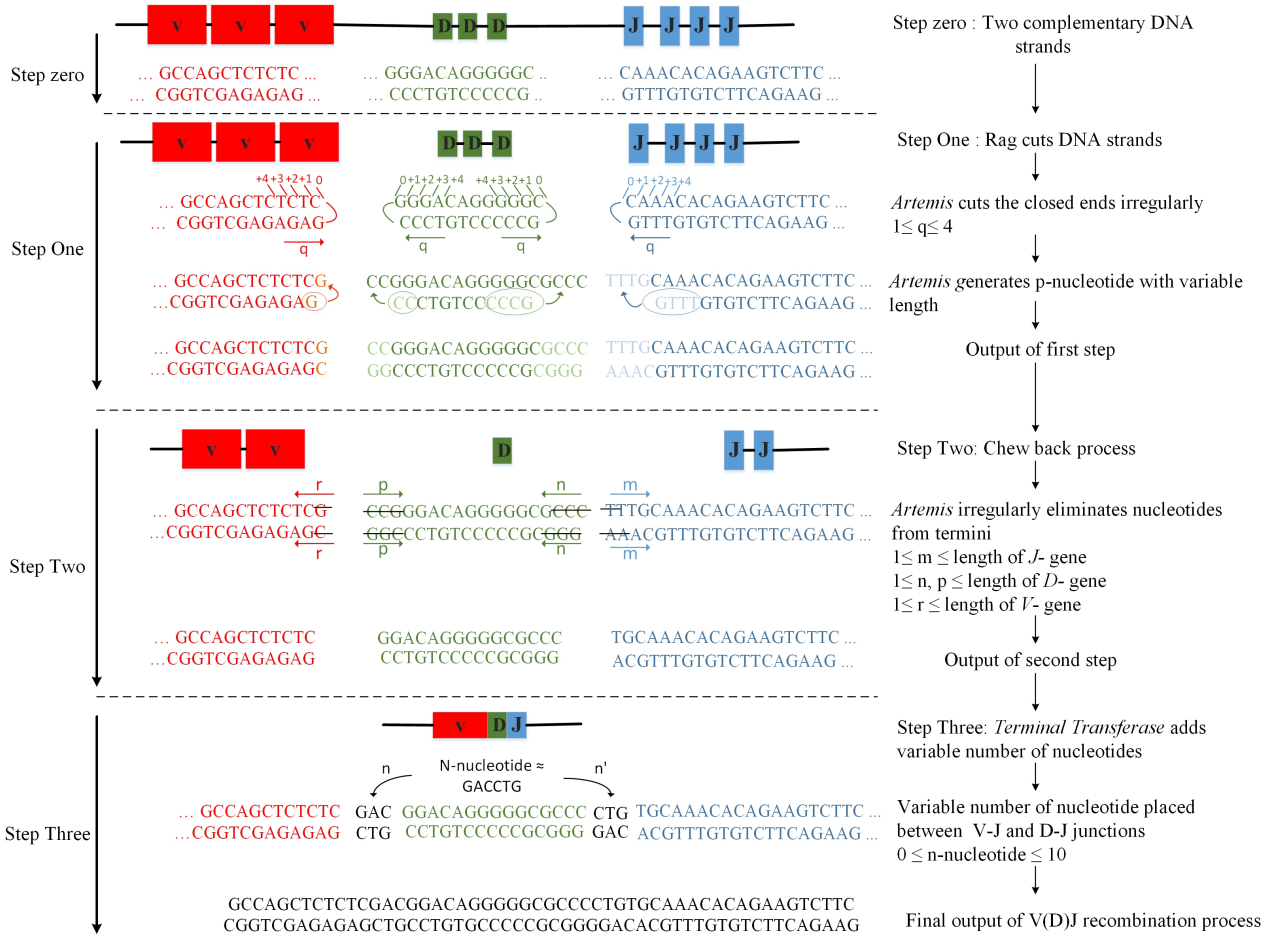

Figure 1: Brief view of the $V(D)J$ recombination process. The figure shows p-nucleotide formation with length one for $V-$ gene termini, two for left side of $D-$ gene termini, four for right side of $D-$ gene termini and four for $J-$ gene termini in step one. Example depicts elimination of one nucleotide on the $V-$ gene termini, three nucleotides on both side of the $D-$ gene and two nucleotides on $J-$ gene termini.

The four nucleotide bases A, G, C, and T are used to generate a nucleotide sequence n. 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 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* sequence. 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.

2.3 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 β genes, 2 D β genes and 12 basic J β 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 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 would result with 18 different sequences based on the chewback process (step two). The D and J gene data sets go through similar process as explained in section 2.1, 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

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

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.

3 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 is the suitable 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 as specified in Section 2.1) along with all possible combinations of four bases for any given n-nucleotide length. 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. 2. As shown, the 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 $V-J$ based assignment results with an uneven workload distribution among the GPU threads.

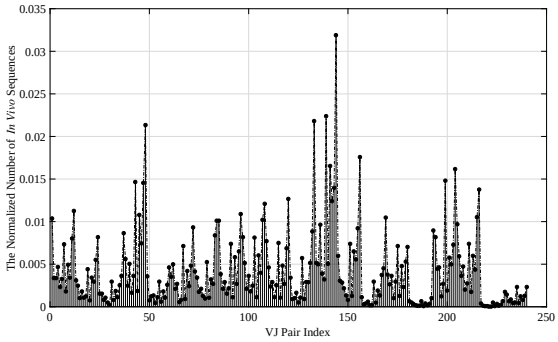

Figure 2: A Normalized distribution of *in vivo* sequences across 240 VJ pairs.

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 the *comparison* step is equal among the GPU threads, since they operate on the same V and J gene. Therefore, total number of *in vivo* sequences for the comparison process are equal among the 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 memory bandwidth utilization.

In order to decide which 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 $V-J$ level parallelism, each thread generates *in silico* sequences for all possible forms of n-nucleotide for length of zero to ten. Table 1 shows the total number of n-nucleotide sequences to generate for each n-nucleotide length from zero to ten. 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 505 sequences for the 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 $V-J$ level parallelism is almost 13 times higher than the n-nucleotide level parallelization approach. Furthermore, from threading power perspective, the n-nucleotide level parallelism offers 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

Table 1: The total number of unique n-nucleotide sequences based on the length of n-nucleotide

n-nucleotide length	*Total number of unique n-nucleotide sequences*
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

$V-J$ level parallelism allows launching only 102,446 threads. Consequently, the n-nucleotide based approach offers higher level of 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. In the following sections we present our approach to bit-wise and multi-GPU implementations based on n-nucleotide level parallelism.

4 BIT-WISE REPRESENTATION

4.1 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 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, 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 refer to the first two bytes, which only contain the original bits of the V sequence, as *full byte*. Note that we store the length of each gene sequence along with the actual gene to distinguish between the *padded 0* and *A*. The length for V, D, and J genes are coded into the first five bits of each gene sequence.

The maximum length of *in vivo* sequences is 60 characters. In the *baseline implementation* [10], *in vivo* sequences are padded with 0 values 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 two 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* [10], 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 implementation allows us to utilize constant memory alone without having to resort to the global memory.

4.2 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 of memory accesses (global or 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. The *baseline implementation* uses this feature to pare down the comparison search space [10]. 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 2 shows the pseudo code for the *task generator* function, which is used to generate a unique binary n-nucleotide sequence. As shown in Algorithm 2, the task generation for each thread involves nested two for loops in which the first for loop iterates through each byte of a n-nucleotide one byte at a time. A n-nucleotide is represented as a three byte package since it can be up to 10 characters long where each is represented using two bits, resulting with a string of 20 bits. The second for loop iterates through the four bases in each byte of n-nucleotide. When these two for loops are unrolled completely, each thread of the GPU is assigned a unique n-nucleotide from all possible n-nucleotides for a given length of *n*. 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 *synchthreads()* 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

Algorithm 2: Pseudo code for the *task generator* function that generates 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] $|=$ $temp$

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 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 the 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 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-nulceotide sequence at any position as explained in section 2. 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 V sequence. The comparison procedure is shown in step two of the Fig. 3, 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-nucleotide sequence, then we load a 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. 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.

5 MULTI-GPU IMPLEMENTATION

In n-nucleotide level parallelization, threads of a single GPU are assigned a unique n-nuclotide 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, block and GPU indexes along with the GPU dimension as shown in (1) and utilize a *task generator* function that is presented in Algorithm 2.

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

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. 4). Finally, we calculate total number of blocks in each GPU using (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 (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.

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

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

6 EXPERIMENTAL SETUP

We conducted our experiments on a cluster consisting of NVIDIA P100 GPU accelerator[6]. 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 2 summarizes the GPU parameters. The P100

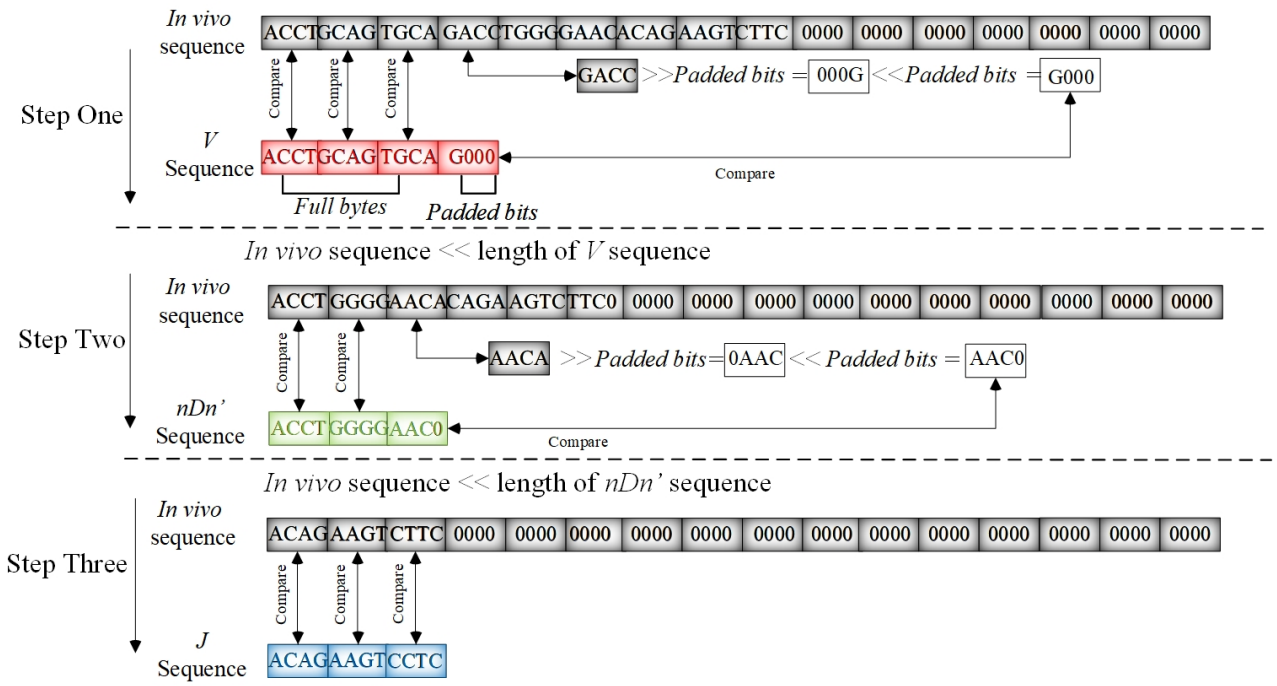

Figure 3: Brief view of the comparison process for the bit-wise version of $V(D)J$ recombination process.

Table 2: 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

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. 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 4, 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.

7 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 implementation* [10] for each n-nucleotide length. Finally, we present execution time analysis for the multi-GPU implementation with up to eight nodes.

7.1 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 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 number of threads to utilize multiple *warps* executing concurrently. As shown in Fig. 4, 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 bellow 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 process 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, 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 $1,365$ active threads per 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 number of 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 result in completing the recombination process in more than one iteration. For 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. However, the number of required iterations 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.

For n-nucleotide length of ten, for all threads per block configurations, the GPU is fully utilized and execution takes the same number of iterations. Therefore we observe negligible difference in terms of execution time among them.

In summary, based on Fig. 4, we set thread block configuration to 64 for 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 with respect to the *baseline implementation*.

Table 3: GPU thread and warp utilization for n-nucleotide length of seven with respect to four different thread block configurations. Values in parenthesis indicate the percentage of utilization.

thread block configuration	*threads per SM*	*thread blocks per SM*	*warp*
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%)

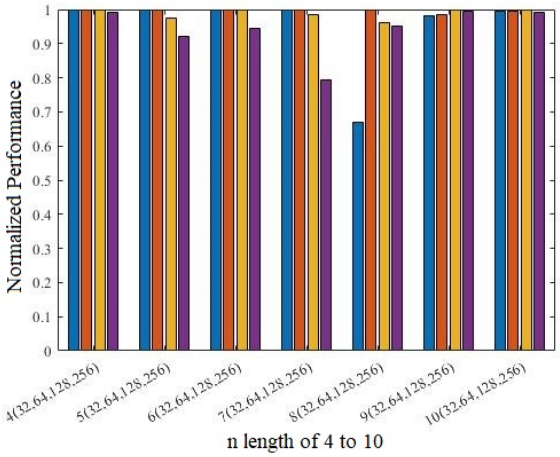

Figure 4: Normalized performance over four threads per block configurations (32, 64, 128, 256) for each n-nucleotide length ranging from four to ten.

7.2 Bitwise Simulation Results

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

Table 4 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 4, 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 5 shows the execution time with respect to the n-nucleotide length from zero to ten for the *baseline* and bit-wise implementations. Last row shows the total execution time for the recombination process. In overall, the total execution time reduces by a factor of 2.1 in comparison with the *baseline implementation*. For n-nucleotide length of eight, we utilize 87.5% of the available SMs on a single GPU with thread block configuration of 64 (shortest execution time). After n-nucleotide of eight, the execution time increases by about a factor of four at each increments of n-nucleotide length by one

Table 4: The memory footprint for the bit-wise implementation in comparison with the baseline approach.

Gene	*Baseline [10]*	*Bit-wise*	*Percentage reduction (%)*
V	1448	425	70.65
J	3107	913	70.61
D	3210	908	71.71
in vivo	6517568	1629392	75.00

Table 5: Execution time on single GPU: Baseline vs. Bit-wise Implementations

N *length*	*Baseline (min)*	*Bit-wise (min)*
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

since SMs become fully utilized and execution turns into iterative flow.

7.3 Multi-GPU Simulation Results

Table 6 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 6 is that there is slight increase in execution time 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 based implementation where execution time is longer compared to single GPU based implementation for n-nucleotide length of up to seven. The reduction overhead is compensated for larger lengths for which we observe reduction in execution time as we increase the number of GPUs.

We observe a reduction in the execution time with multiple GPUs for n-nucleotide lengths that are beyond seven. This is due to the fact that a single GPU is almost fully utilized at 87.5% for n-nucleotide length of more than seven as explained in section 7. 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 lengths 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 trend in execution time reduction where adding another GPU resource no longer helps reduce the execution time. 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 ($ceil(4^9/1280 \times 56)$) 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 threads on both GPUs. 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 ($ceil(4^{10}/1280 \times 56)$) using a single GPU while there are 45, 056 active threads in the last iteration (62% GPU threads utilization). However, employing two GPUs results in completing the process in eight 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 five iterations while the GPU thread utilization is at 87.6% in the last iteration. When we employ four GPUs, iteration cont becomes four. For the GPU count of five, six, and seven, the iteration count remains at three 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 6, 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 6, the execution time for n-nucleotide length of nine is 196 minutes based on single GPU.

Table 6: Execution time in minutes for each nucleotide length (0 to 10) with respect to number of GPUs (1 to 8) for the bit-wise implementation.

N length	*1-GPU (min)*	*2-GPUs (min)*	*3-GPUs (min)*	*4-GPUs (min)*	*5-GPUs (min)*	*6-GPUs (min)*	*7-GPUs (min)*	*8-GPUs (min)*
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

However we observe that execution time as 231 minutes for n-nucleotide of ten with four GPUs. We identify three factors for this discrepancy. First one is the overhead of reduction process with the increase in number of GPUs that was explained earlier in this section. Second factor is the difference between the total number of nDn' combinations. As stated earlier in Section 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 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. 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 4 GPUs. This discrepancy is about 2.6 minutes for the length pair of seven and eight with 1 and 4 GPUs respectively. We also attribute this discrepancy reduction trend to the three factors listed above.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce bit-wise implementation of the $V(D)J$ recombination algorithm, which reduces the constant memory and global memory footprint by factors of 3.4× and 4× respectively. On a single GPU, the bit-wise implementation reduces the total execution time by a factor of 2.1× compared to the baseline implementation. We then present the multi-GPU version of the bit-wise recombination and conduct scalability analysis. We show that beyond n-nucleotide of eight, since we fully occupy the thread blocks on a single GPU, we observe reduction in execution time with the increase in number of GPUs. However this reduction shows a saturating trend. We finally analyze the root causes of observing a saturation tend 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 that the baseline version.

REFERENCES

[1] Mark M. Davis, J. Jay Boniface, Ziv Reich, Daniel Lyons, Johannes Hampl, Bernhard Arden, and Yueh hsiu Chien. 1998. Ligand recognition by $\alpha\beta$ T cell receptors. *Annual Review of Immunology* 16 (April 1998), 523–544. https://doi.org/10.1146/annurev.immunol.16.1.523

[2] George Du, Crystal Y. Chen, Yun Shen, Liyou Qiu, Dan Huang, Richard Wang, and Zheng W. Chen. 2010. TCR repertoire, clonal dominance, and pulmonary trafficking of mycobacterium-specific CD4+ and CD8+ T effector cells in immunity against tuberculosis. *Journal of Immunology* 185 (Oct. 2010), 3940–3947. https://doi.org/10.4049/jimmunol.1001222

[3] Martin Gellert. 2002. V(D)J Recombination: RAG Proteins, Repair Factors, and Regulation. *Annual Review of Biochemistry* 71 (July 2002), 101–132. https://doi.org/10.1146/annurev.biochem.71.090501.150203

[4] Jorge MansillaSoto and Patricia Cortes. 2003. V(D)J recombination: artemis and its in vivo role in hairpin opening. *The Journal of Experimental Medicine* 197, 5 (April 2003), 543–547. https://doi.org/10.1084/jem.20022210

[5] Lieber MR. 1991. Site-specific recombination in the immune system. *The Journal of the Federation of American Societies for Experimental Biology* 5 (Nov. 1991), 2934–2944.

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

[7] NVIDIA 2018. NVIDIA CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[8] Yu Ping, Chaojun Liu, and Yi Zhang. 2018. T–cell receptor–engineered T cells for cancer treatment: current status and future direction. *Protein and Cell Journal* 9 (March 2018), 254–266. https://doi.org/10.1007/s13238-016-0367-1

[9] David G. Schatz and Yanhong Ji. 2011. Recombination centers and the orchestration of V(D)J recombination. *Nature Reviews Immunology* 11, 4 (April 2011), 251–263. https://doi.org/10.1038/nri2941

[10] Gregory Striemer, Harsha Krovi, Ali Akoglu, Benjamin Vincent, Ben Hopson, Jeffrey Frelinger, and Adam Buntzman. 2014. Overcoming the limitations posed by TCRβ repertoire modeling through a GPU-Based in-silico DNA recombination algorithm. *in Parallel and Distributed Processing Symposium,* 28th IEEE Int. parallel and distributed processing symp (May 2014), 231–240. https://doi.org/10.1109/IPDPS.2014.34

[11] TCR Synthesis 2019. TCR Synthesis CUDA implementation. http://www2.engr.arizona.edu/~rcl/publications/source_codes/recombination/TNTSoftware.zip

[12] Benjamin Vincent, Adam Buntzman, Benjamin Hopson, Chris McEwen, Lindsay Cowell, Ali Akoglu, Helen Zhang, and Jeffrey Frelinger. 2016. iWAS–A novel approach to analyzing next generation sequence data for immunology. *Cellular Immunology* 299 (Jan. 2016), 6–13. https://doi.org/10.1016/j.cellimm.2015.10.012

[13] Raymond M. Welsh, Jenny W. Che, Michael A. Brehm, and Liisa K. Selin. 2010. Heterologous immunity between viruses. *in Immunological Reviews Journal* (April 2010), 244–266. https://doi.org/10.1111/j.0105-2896.2010.00897.x

[14] Agustin Zapata and Chris Amemiya. 2000. Phylogeny of lower vertebrates and their immunological structures. *Current Topics in Microbiology and Immunology* 248 (Oct. 2000), 67–107. https://doi.org/10.1007/978-3-642-59674-2_5