Overcoming the Limitations Posed by TCRβ Repertoire Modeling through a GPU-based In-Silico DNA Recombination Algorithm

Gregory Striemer*, Harsha Krovi†, Ali Akoglu*, Benjamin Vincent§, Ben Hopson¶, Jeffrey Frelinger‡ and Adam Buntzman‡
*Department of Electrical and Computer Engineering University of Arizona,Tucson, AZ 85721
Email:gmstrie@email.arizona.edu, akoglu@ece.arizona.edu
† Integrated Department of Immunology University of Colorado, Denver , CO, 80206
Email: shk8f@virginia.edu
‡Department of Immunobiology University of Arizona, Tucson, AZ, 85719
Email: buntzman@email.arizona.edu,jfrelin@email.arizona.edu
§Department of Medicine, Division of Hematology-Oncology University of North Carolina-Chapel Hill, NC, 27599
Email: BGVincen@unch.unc.edu
¶School of Engineering, The University of Edinburgh, Edinburgh, EH9 3JL, UK
Email: B.Hopson@ed.ac.uk

***Abstract*—The DNA recombination process known as V(D)J recombination is the central mechanism for generating diversity among antigen receptors such as T-cell receptors (TCRs). This diversity is crucial for the development of the adaptive immune system. However, modeling of all the α β TCR sequences is encumbered by the enormity of the potential repertoire, which has been predicted to exceed 10^{15} sequences. Prior modeling efforts have, therefore, been limited to extrapolations based on the analysis of minor subsets of the overall TCRβ repertoire. In this study, we map the recombination process completely onto the graphics processing unit (GPU) hardware architecture using the CUDA programming environment to circumvent prior limitations. For the first time, we present a model of the mouse TCRβ repertoire to an extent which enabled us to evaluate the Convergent Recombination Hypothesis (CRH) comprehensively at peta-scale level on a single GPU.**

***Keywords*-cuda; dna; recombination; gpu; t-cell receptors;**

I. Introduction

Immune systems of jawed vertebrates depend heavily on the diversity of their antigen receptors (immunoglobulins and T cell receptors (TCRs)) [1]. The diversity itself is created via a specialized DNA recombination process known as V(D)J recombination [2], which is the central mechanism behind the generation of the vast numbers of receptors in the adaptive immune system. The TCR is composed of an α and β chain (TCRα and TCRβ) [2], each of which undergoes recombination, that involves rearrangement of variable (V), diversity (D), and joining (J) gene segments chosen from members of each gene family [2], [3]. Successful rearrangement is crucial for antigen recognition and hence, for the adaptive immune system. Combinations of the V, D, and J gene segments enable construction of numerous unique receptors. Overall, the size of the total possible $\alpha\beta$ TCR repertoire in mice has been estimated to exceed 10^{15}, with over $5x10^{12}$ unique TCRβ chains [6], [7].

Until now, immunologists have been struggling to predict the outcomes of immune responses [9], [10], [22], [23] to foreign antigens such as vaccines, since this process requires modeling of all of the $5x10^{12}$ possible TCR sequences (*in silico*) and then counting the number of times each sequence is re-made by different recombination paths. According to the major accepted hypothesis (Convergent Recombination Hypothesis [16], [17]), such modeling would allow us to define the subset of TCRs that would have the highest probability of participating in immune responses. Hence, a pressing need exists to model the recombination process with affordable high performance computing solutions. Reducing the time it takes to extract this information would allow immunologists to study fundamental questions such as: How large can the repertoire truly be? Is the establishment and maintenance of the TCRβ repertoire random or biased? If biases exist, what controls the frequency distribution of the species that are generated in the TCRβ repertoire? Why do we see some TCRβ clones ("public clones") reappear in many individuals, when their observance across individuals is statistically unlikely?

Prior attempts at modeling the TCRβ repertoire in mice have been limited to the analysis of a minor subset of the total TCRβ repertoire followed by extrapolation to the global repertoire [11], [12], [13], [14], [15], [16], [17]. Utilizing a graphics processing unit (GPU) allowed us to circumvent this major limitation by providing a platform where the global repertoire could be modeled more extensively. We identified ways of structuring the recombination process from resource utilization and performance perspectives such that the program architecture overlaps with the GPU. We exploit the memory hierarchy of the GPU architecture for data movement between registers, caches, shared memory and external DRAM for hiding memory latencies. Inner

and outer loop modifications are employed to make the process suitable for n-way SIMD based implementation and to expose task level parallelism, which improves code scheduling by enlarging the block size. To eliminate the need for communication between GPU threads, we introduce a function that allows each thread to generate a unique "n" nucleotide based on its thread identification number. We evaluate the impact of launching threads in blocks. Based on our exhaustive fine tuning efforts for identifying the most suitable number of threads per block with various workloads, we carefully balance the data partitioning among threads, form thread counts that are factors of the workload, have an even workload distribution among all multi-processors, and avoid idling threads whenever possible. We completely map the recombination process on the GPU and validate its functionality based on the one-to-one output match with the sequential algorithm outlined previously [16], [17].

To support our in-silico modeling, we first sequenced the global TCRβ repertoire of two mice using a novel genomics-based technique of ultradeep sequencing. Next, we enumerated the ways in which each *in vivo* TCRβ sequence could be synthesized *in silico*. For the first time, we present a model of the mouse TCRβ repertoire that synthesizes TCRβ sequences on the order of $4x10^{14}$ recombination pathways [18]. Specifically, due to fine-grained thread-level parallelism offered by the GPU architecture, we managed to count the recombination paths of sequences at a larger scale ($1x10^{8}$ fold higher than prior studies). This level of *in silico* TCRβ analysis enables immunobiologists to interrogate a process that was heretofore untenable.

The remainder of this paper is organized as follows: We describe the V(D)J recombination process from biological and algorithmic perspectives in section 2. We outline this process from algorithmic point of view, derive the formula for the total number of recombination paths, and quantify size of the recombination search space. This will set the stage for section 3 on parallelization strategies employed and design decisions made for mapping the recombination process onto the GPU architecture. We present our evaluation strategy in section 4, followed by performance analysis in section 5. Here we also discuss the significance of our findings from an immunological standpoint. Finally, we present our conclusions in section 6.

II. VDJ Combinatorial Diversity

V-, D- and J-genes are juxtaposed with each other by numerous enzyme complexes during rearrangement to form the receptors [1], [2], [7]. As depicted in figure 1, *RAG-recombinase* first brings a D-gene next to a J-gene and circularizes both their ends. Subsequently, *Artemis exonuclease* liberates the circular ends promiscuously to generate a variable number of P-nucleotides. *Artemis exonuclease* is once again recruited to the junctions to variably remove nucleotides from the D- and J-gene termini. To increase the junctional diversity of these receptors, *Terminal Transferase (TdT)* is recruited to the junctions to add any of the four nucleotides in a template-independent fashion. Finally, *DNA ligase IV* joins the modified D-gene and J-gene to produce a complete DJ junction. This very process is repeated with a chosen V-gene to create the full VDJ recombinant.

Attempts have been made to interrogate the TCR repertoire by sequencing TCRs and modeling the TCR synthesis *in silico*. Thus far, these approaches have been minimalist since they have omitted the inclusion of P-nucleotides in the modeling and have simply analyzed a small subset of the possible repertoire. For example, prior computational models have randomly sampled a small pool (10^{6}) of TCRs from an *in silico* meta-repertoire whose parameters allow for over 10^{15} recombinants [17]. The random sampling inherently biases the selection of TCRs with enormous n-addition parameters. This conclusion is supported in [16] by Venturi et al where they note that random sampling is "effectively biasing the simulation toward producing a greater proportion of sequences with a high number of nucleotide additions than demonstrated by the distributions". Such approaches are applied due to the complexity and enormity of the total repertoire, and the lack of tenability to model the whole repertoire on classical computational systems. Ideally, though, all of the possible recombination paths to all possible TCRs should be generated, and the recombination paths to each TCR sequence enumerated. Among this vast set, identifying all possible recombination paths that generate each TCRβ sequence would allow us to define the subset of TCRs that have the highest probability of being generated *in vivo*, according to the CRH. Furthermore, this allows for the prediction of which sequences would be present at a high frequency *in vivo*. As a consequence, this may enable researchers to rationally predict those TCRs that participate in a protective immune response or those that may participate in autoreactive immune processes.

A. Algorithm

In this subsection, we will describe the recombination process from an algorithmic vantage point. Any sequence which we generate will be referred to as *in silico*. The inputs needed to perform recombination include a set of V-, J-, D-, and n-nucleotide sequences of various lengths. All germline termini participating in recombination can include up to 4 palindromic nucleotides (p-nucleotides). The *in vivo* sequences will later be compared to the *in silico* generated sequences. The objective is to find the number of times each *in vivo* sequence can be artificially generated *in silico* using the input sequences. For the *in silico* recombination, we will refer to V(D)J recombination as VnDn'J recombination. Here, V, D, and J represent unique sequences from each set, and "n" represents the set of all the possible nucleotide combinations, with the restriction that the length of each combination is between 0 and 10 (inclusive). Given the four

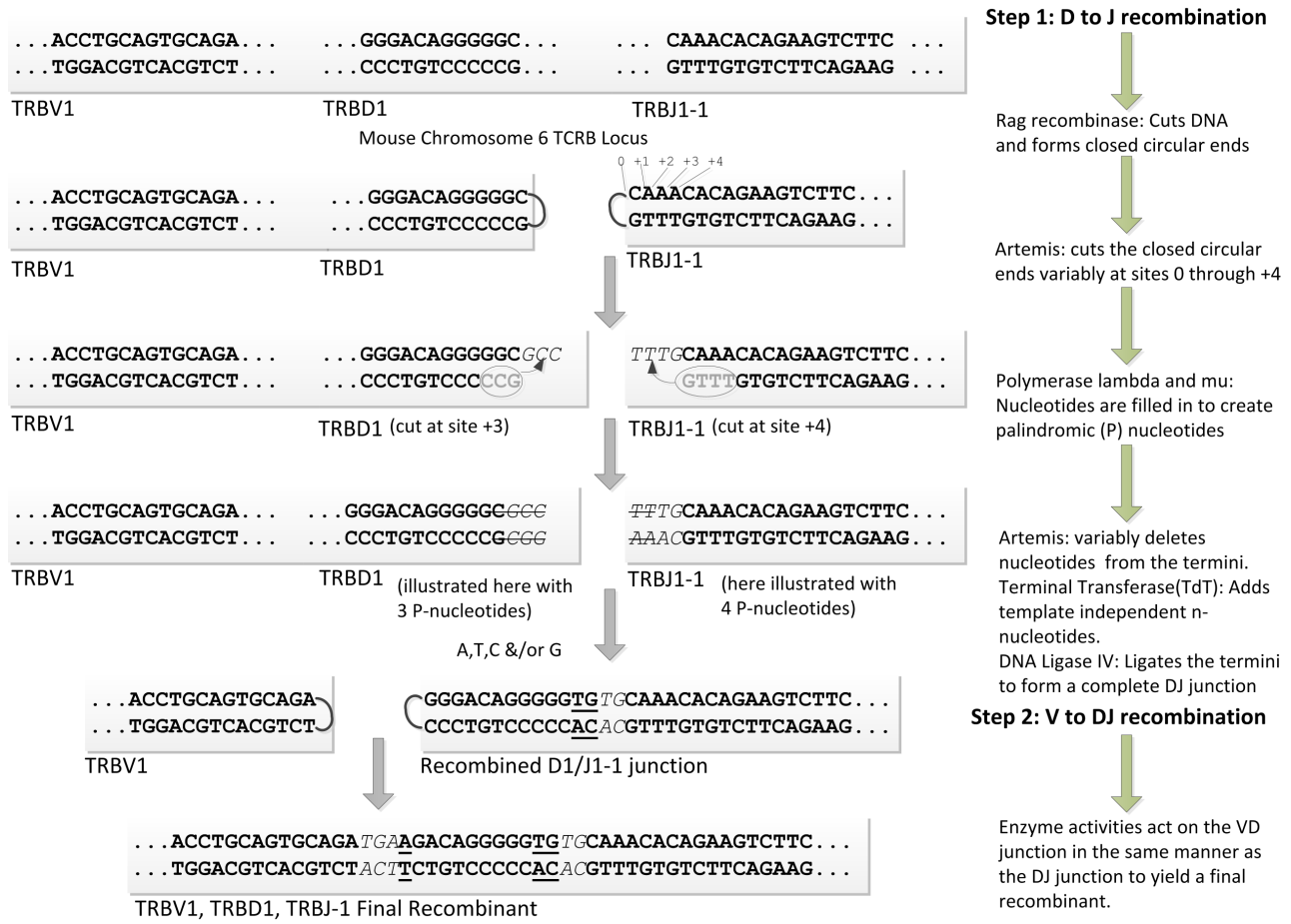

Figure 1. Overview of the overall V(D)J recombination process as it occurs for the TCRβ chain. This figure depicts Artemis hairpin-liberating and exonuclease activities, P-nucleotide formation, and TdT template-independent nucleotide addition, among others. Example V-, D-, and J-genes are used to illustrate the process. P-nucleotides are italicized and not bolded while n-nucleotides are underlined.

nucleotide bases (A, G, C, T), for n-nucleotide sequences of length m, there can be 4^m nucleotide combinations. These n-nucleotides can occur exclusively on one side of the D-sequence or on both sides of the D-sequence, which we define as n and n'. The D-sequence can divide the n-sequence at any position. This junctional complexity ensures that there will be numerous ways of recreating an identical sequence during the recombination process. In C57BL/6 mice, there are 20 functional mouse Vβ genes, 2 Dβ genes, and 12 functional Jβ genes, where the Dβ1 gene can pair with all 12 genes of the Jβ1 and Jβ2 gene cassettes (240 combinations), and the Dβ2 gene can only pair with the 6 genes of the Jβ2 gene cassette (120 combinations). Here we note that the actual V and J genes are specific to a given species. For example, humans will have a different set of V and J sequences than the mice. The combinatorial diversity with just these parameters would allow for only 360 different combinations of V, D, and J genes. However, incorporating 4 P-nucleotides at the ends of the genes and the variable nucleotide deletions generate 362 possible Vβ gene termini, 230 paths to all the Dβ1 species, 275 paths to the Dβ2 species, and 283 possible Jβ gene termini (144 termini for the Jβ1 cassette and 139 termini for the Jβ2 cassette). The total combinatorial diversity generated by inclusion of 4 P-nucleotides and the exonuclease digestion of the gene ends yields 37,400,030 possible recombination paths based on equation 1. This implementation does not address the limiting biological case where P-nulceotides are liberated and subsequently removed by Artemis.

$$
\begin{aligned}
&RecombinationPaths = \\
&\quad \sum V_{length} * J_{length} * \\
&\quad (4^n * (n+1) * (DB1_t - DB1_e) + 4^n * DB1_e)) + \\
&\quad V_{length} * J_{length} * \\
&\quad (4^n * (n+1) * (DB1_t + DB2_t - DB1_e - DB2_e) + \\
&\quad (4^n * (DB1_e + DB2_e))) \qquad (1) \\
&\quad where \\
&\quad 0 \leq V_{length} \leq 16 \text{ - } 19, 0 \leq J_{length} \leq 21 \text{ - } 27
\end{aligned}
$$

Table I
THE TOTAL POSSIBLE NUMBER OF COMPARISONS BETWEEN THE IN SILICO GENERATED SEQUENCES AND IN VIVO DATASET.

In Silico Comparisons		
$\mathbf{n_{len}}$	$\mathbf{n_{comb}}$	**Total Comparisons**
0	1	$13,704,874,784$
1	4	$109,354,392,096$
2	16	$655,557,140,224$
3	64	$3,494,786,848,256$
4	256	$17,469,380,542,464$
5	$1,024$	$83,838,454,767,616$
6	$4,096$	$391,197,549,461,504$
7	$16,384$	$1,788,165,119,410,176$
8	$65,536$	$8,046,160,163,897,344$
9	$262,144$	$35,758,639,400,615,936$
10	$1,048,576$	$157,330,552,582,569,984$

for $i = 0 \rightarrow NumberOf_{Vseq}$ **do**
  for $j = 0 \rightarrow NumberOf_{Jseq}$ **do**
    for $k = 0 \rightarrow NumberOf_{Dseq}$ **do**
      for $m = 0 \rightarrow NumberOf_{n-nucleotideseq}$ **do**
        for $t = 0 \rightarrow NumberOf_{In-VivoSequences}$ **do**
          $Combination = CombineString(V[i], N[m], D[k], J[j])$
          for $p = n - nucleotide_{length=m} \rightarrow 0$ **do**
            if $Combination == I[t]$ **then**
              $Hit_{count}[t] += 1$
            end if
            $Move(N[m][p] \rightarrow T[n - nucleotide_{len} - p])$
          end for
        end for
      end for
    end for
  end for
end for

Figure 2. Basic outline of the serial recombination process and the recombination pathway enumerations for each in vivo sequence.

$$DB1_t = 230, DB1_e = 20, \text{ t: total, e:empty}$$
$$DB2_t = 275, DB2_e = 22$$
$$0 \leq n \leq 10$$

Additional diversity is generated by the non-templated enzymatic addition of n-nucleotides at the V-D and D-J junctions. Upon including a maximum of 10 n-nucleotides, the number of possible recombination paths increases substantially to 398,292,334,673,920. Table I illustrates the total possible sequence comparisons between the *in vivo* and *in silico* datasets.The pseudocode shown in figure 2 provides a high-level view of the VnDn'J recombination process. In this example, "I" refers to a set of *in vivo* sequences. To ensure that every possible VnDn'J combination is exhaustively produced, there are nested loops that iterate through each V, D, J and n sequence. Within these loops, the individual sequences are combined and stored in the variable $Combination$. If the produced sequence matches the current *in vivo* sequence, then we increment the counter (Hit_{count}). If there is no match, then the n-nucleotide sequence is moved one character at a time to the *n'* position. Each time a character is moved to the *n'* position, the newly created sequence is compared afresh against the *in vivo* set.

III. GPU BASED RECOMBINATION

The recombination algorithm presented by Davenport in [16], [17], would have led to prohibitive execution times if all recombination paths were enumerated. Therefore a random sampling approach was used to choose what sequences were actually selected from the *in silico* space of the model. Only a small portion of the randomly selected *in silico* sequences actually match the biological sequences, so the sampling comparison between the *in silico* and *in vivo* datasets is small. Davenport's method relies on extrapolating the number of recombination paths from the number of times that an *in silico* sequence is randomly chosen from the *in silico* program, instead of counting the number of recombination paths. We have followed Davenport's algorithm with the addition of modeling *p-nucleotide* additions and improved on it by 1) enumerating all of the recombinations that would yield each sequence that was observed within our biological dataset; and 2) recording the total number of recombination paths to each sequence without any extrapolation.

The GPU programming approach taken in this work attempts to write a functionally equivalent algorithm presented previously [16], [17]. Such an implementation was designed to optimally utilize the threading power offered by GPUs. We employ strategies for early termination during recombination when we know a sequence will not match any sequence in the *in vivo* set. This is accomplished not by building an entire sequence and then comparing it to the *in vivo* set, but rather comparing as it is built. We used two arrays, with one containing the *in vivo* sequences and the other containing a set of indexes for the *in vivo* array. This indexing array allows us to determine where each *in vivo* sequence begins and ends. This reduces our search time to $O(1)$ for matching an *in vivo* and *in silico* pair because we are always aware of the location of the sequences we wish to compare. Rather than storing entire *in silico* sequences during recombination, we hold only a single character from V, J, D, or n-nucleotide sequences at any given time. This reduces the memory footprint from having to store all *in silico* combinations to only a single character per thread.

The host side of the program uses the V, D, and J sequence files as inputs along with an index file that contains *in vivo* sequences within each VJ pair. The V, D, J, and index information is stored in the GPU's constant memory so that we can take advantage of the temporal locality it offers. However, since there are too many *in vivo* sequences to fit in the constant memory ($> 100K$), they must be stored in the GPU's global memory. Each sequence is fewer than 64 characters in length; therefore, we pad the ends of each

$base[4] \leftarrow \{A, T, G, C\}$
$G_{tid} \leftarrow blockID * blockDim + blockThreadID$
$n[0] \leftarrow base[Gtid\%4]$
for $i = 1 \rightarrow n - sequence_{Length} - 1$ **do**
  $n[i] \leftarrow base[(G_{tid} + \frac{G_{tid}}{4^i})\%4]$
end for

Figure 3. Method to ensure that each thread is assigned a unique n-nucleotide sequence during the recombination process.

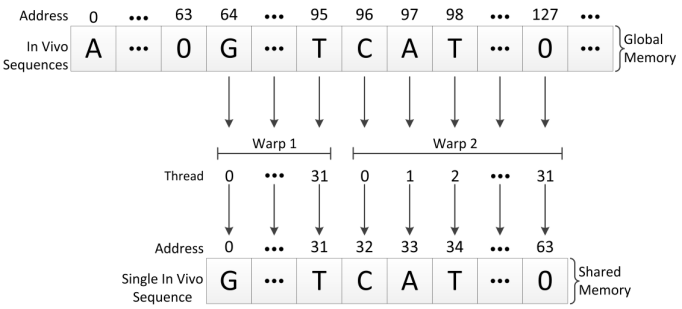

Figure 4. Reading an *in vivo* sequence from global memory with two warps. Padding of zeros is used to ensure each sequence length matches the number of threads in two warps.

sequence with 0's so that the lengths of all sequences are equal to 64. This ensures that the allocation of each sequence will be equal in size to the number of threads in two warps, and allows us to easily coalesce our reads, while also ensuring the memory is aligned.

For our design, the number of threads is entirely dependent on the length (*m*) of the n-nucleotide sequence. We set the number of threads equal to the total possible combinations for a given n-sequence length (4^m). This is convenient because it is always a power of 2. When the n-nucleotide sequence has a length greater than 3, number of threads will always be evenly divisible by the size of a warp.

The index input file briefly discussed earlier helps separate the *in vivo* sequences by their VJ pairs, yielding 240 different groups. Consequently, this allows us to pare down the *in vivo* search space we must mine for *in silico* matches. There are two loops on the host side in which the GPU kernel call is enclosed. Combined, they iterate through each of the V and J sets. There are a total of 20 V sets, or files, and a total of 12 J sets. Upon being launched, a kernel uses the *in vivo* index file and the current VJ pair to compare only the pertinent *in vivo* sequences against the generated sequences. After the kernel completes its execution, the output will be a list of the number of times each *in vivo* sequence was created during recombination. However, each thread-block will output its own set of numbers for each *in vivo* sequence. Therefore, these values are condensed for each *in vivo* sequence with a simple reduction on the host side.

A. GPU Kernel

The GPU begins by determining how many, and which, *in vivo* sequences will be compared based on which VJ sets are used to perform the *in silico* recombination. Each thread in the kernel is then assigned a unique n-sequence combination. Since there are as many kernel threads for a given n-sequence length as there are n-sequences, we must ensure that there are no duplicate n-sequences. To accomplish this, we developed a function which could quickly assign a unique n-sequence combination to each thread given the length of n-sequence, a thread ID, a block ID, and thread-block dimensions (figure 3). Because the length of n-sequence is small (0 to 10), the loop is completely unrolled.

After each thread determines its own n-nucleotide sequence, we start the recombination process on the GPU. There are four main loops in the GPU code. These iterate through the V, D, J, and *in vivo* sequences. The first loop iterates through the *in vivo* sequences. Upon entering this loop, we read a single *in vivo* sequence from the global memory into the shared memory. Threads within a thread-block cooperate in this process to coalesce the memory read. Since we added padding to ensure that each *in vivo* sequence occupies 64 bytes (characters) in the global memory, the first two warps of each thread-block receive an *in vivo* sequence as input. The process of reading from the global memory to the shared memory is illustrated in figure 4. Since this data will be shared among all threads within a thread-block, we added the *synchthreads()* function in the code to ensure all threads within the thread-block wait until the memory transaction is complete before proceeding.

The second loop iterates through each V sequence in the current set. While each thread operates on a different n-nucleotide sequence, all threads read the same V sequence from the constant memory for recombination. Therefore, we are able to make use of the temporal locality of the constant cache. *In Vivo* sequences are stored one at a time in the on-chip shared memory. The entire sequence is indeed stored in the shared memory. Since we perform recombination to look for exact matches with the *in vivo* sequences, we handle this one character at a time. If a single character is a mismatch, then we load a new *In Vivo* sequence into the shared memory without having to combine D, n, or J sequences to that particular V sequence. When comparing characters from the *in vivo* sequence, each thread reads the same character within a thread-block. This allows us to broadcast that character from the shared memory to all threads. If the V sequence does match the first part of the *in vivo* sequence, then we proceed to the next loop.

The third loop iterates through all of the D sequences. Again, if there is a single character mismatch we terminate the current comparison. However, the D-loop differs from the V-loop in that when we have a mismatch, we load a new D sequence, not a new *in vivo* sequence. Also, since the n-sequence characters can be placed on either side of

the D sequence, we combine the thread-resident n-sequence to the D-sequence in this loop. Because of this, for each D-sequence with a length greater than 0, we have $4^m * (m+1)$ combinations. However, when the D-sequence length is 0 (i.e., when the D-sequence has been completely chewed back by Artemis exonuclease), the n-nucleotide sequence can only be placed in one location and the number of combinations reduces to 4^m. We compare each of these amalgamated sequences one character at a time from the last character that was found to be identical to the V-sequence in the previous loop. If there is a match to the *in vivo* sequence thus far, we continue on to the fourth and final loop.

The final loop iterates through the set of J sequences and compares them one character at a time to the latter portion of an *in vivo* sequence. Before actually performing a character-by-character comparison, we perform a quick litmus test at this point. Since we know the length of the particular V and nDn' sequences thus far, we determine the total length of the recombination sequence by including the length of a J sequence within the current loop. To quickly determine if we have a possible match, we compare the length of VnDn'J to the length of the *in vivo* sequence. If it is not the same length, then we proceed in the loop to operate on a new J sequence. If it is a match, then we begin our character-by-character comparison to the *in vivo* sequence. Performing this check is advantageous, because it is far more likely we will have a mismatch than a match. If a sequence completely matches the *in vivo* sequence, then the thread corresponding to that sequence increments the counter. This counter is stored in a register, and then transferred to the shared memory. At the end of loop four, we perform a reduction at the granularity of a thread-block. In this reduction, we determine the total number of times each thread was able to successfully create an *in vivo* sequence. The first thread in each thread-block then writes this reduced value to the global memory.

IV. Evaluation Strategy

Splenic $CD8^+$ T-cells were magnetically isolated (using Miltenyi Biotec $CD8^+$ T-cell negative isolation reagent) from two female 8-week old C57BL/6 mice. Purity was shown to exceed 93% by flow cytometry. Samples were sequenced on the Roche FLX 454 platform at the UNC-Chapel Hill High Throughput Genome Sequencing Core. We were able to retrieve 101,822 unique, functional sequences, which were then utilized for all further informatics evaluation. The germline Vβ, Dβ, and Jβ gene sequences and nomenclature were all obtained from IMGT [20].

Upon retrieving the sequences from the two mice, we first implemented the recombination algorithm previously introduced in [16], [17]. We use this baseline implementation as a reference code for verifying the functionality of the GPU code. Using both the baseline and the CUDA programs, we modeled all possible TCRβ sequences and enumerated the number of times each *in vivo* sequence is re-made by different recombination pathways for n-nucleotide lengths from 0 to 7. Both programs resulted in a one-to-one match. Source files for the baseline and CUDA implementations along with the datasets are available as a package [18].

The baseline code is single-threaded and written in Python. It is executed on the 2.83GHz quad-core Intel Xeon processor with 2GB RAM/core. During the GPU execution time analysis, we will refer to the execution time of the baseline code just to set the stage for our arguments on the computational challenge the recombination generation process poses, and illustrate the need for alternative affordable high performance computing platforms. In this paper, our intention is not to compare the performance of GPU implementation against the Python code, but rather to show that the GPU based implementation enables modelling of the whole mouse TCRβ repertoire to an extent where we are able to appropriately evaluate the Convergent Recombination Hypothesis (CRH) and gain insight into the biases. We have tested the CUDA based implementation on an NVIDIA GTX 480 GPU operating at 1.4 GHz with a host machine containing an Intel Xeon quad-core processor.

The GPU consists of an array of 15 Multiprocessors (MPs) each with 32 processing cores. A MP is capable of maintaining a maximum of 8 active thread-blocks at any given time. During kernel execution, threads within a thread-block are broken down into groups of 32, called *warps*. The maximum number of resident warps per multiprocessor is 48. The fastest memories are on-chip registers and shared memory, with each taking as little as 1 clock cycle to access. Each multi-processor contains 32,768 32-bit registers and 48KB of shared memory, which are distributed evenly among all active thread blocks. Each multiprocessor is capable of having up to 1536 active threads.

We successfully modeled the TCRβ recombinations up to a n-nucleotide addition of 10 on the GPU. We used 10 as the upper bound since the cumulative coverage (0 to 10) is over 99.5% of the *in vivo* dataset. We illustrate the cumulative coverage trend with respect to the length of the n-nucleotide in figure 5. Another reason for modeling up to a n-addition of 10 nucleotides was to ensure that our *in silico* model was not unintentionally biasing the number of recombination paths to be greater in one subset than another [16], [19]. Similarly, our rationale for modeling up to 4 P-nucleotides was founded on both previous studies as well as our *in vivo* data. Prior studies have outlined the extent of P-nucleotides generated by Artemis upon hairpin liberation. These findings conclude that when P-nucleotides are present, there are primarily 1-4 of them appended to the ends of the gene segments [7]. Corroborating this result, in our *in vivo* dataset, we are able to regenerate over 99.5% of the sequences by employing up to 4 P-nucleotides. In table II, we present a side-by-side comparison between our *in silico* model and the prior *in silico* model implemented by Quigley et al [17].

The recombination process is completely executed on the

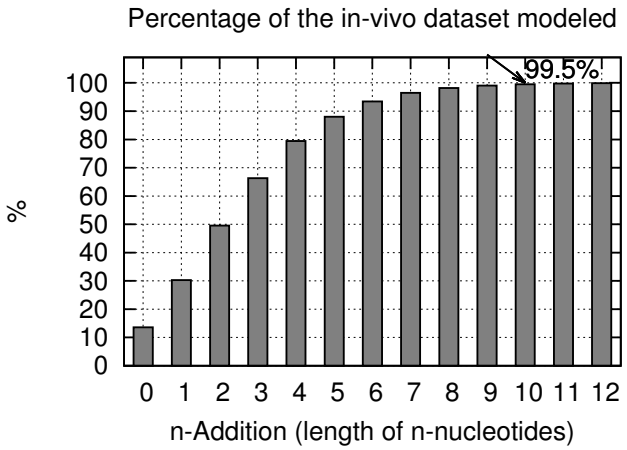

Figure 5. Cumulative coverage of *in vivo* dataset over n-nucleotide length.

Table II
COMPARISON BETWEEN THE *in silico* MODEL PRESENTED HERE AND PREVIOUS *in silico* MODELS.

Feature	**Prior Models**	**Our Model**
V Gene Nucleotide Deletion	0-10 or 12	All possible
J Gene Nucleotide Deletion	0-10 or 12	All possible
D Gene Nucleotide Deletion Paths	Limited to species	All possible
Palindromic "P" Nucleotides	–	$0-4$
Non-templated "n" Nucleotides	0-10 or 12	$0-10$
All functional VJ combinations	Only one VJ pair	All possible
Enumerating all possible paths	*Random*	All possible

GPU. We use execution time for the entire program as the performance metric, which also includes reading in data files, all memory transfers between the host and device, and writing the final results to a file. We first determine the best possible thread configuration based on the execution time of several test runs, and then evaluate the effectiveness of our GPU implementation by measuring the hardware thread-block and warp utilization for each n-nucleotide length. We present the GPU execution times for enumerating all possible recombination paths to the *in vivo* derived TCRβ sequences for up to a length of 10 n-nucleotide addition. For the baseline code, we report execution time sequences for up to 7 n-nucleotide addition as it is not feasible to run the recombination process on a single core beyond this point because of its extremely long execution time.

V. EXPERIMENTAL RESULTS

We ran experiments with different thread-block configurations in order to determine the best possible thread organization for different n-nucleotide lengths. In figure 6, we illustrate the normalized results for n-nucleotide lengths from 3 to 10. We ignore 0 through 2 as there are not sufficient threads to require multiple warps running concurrently. We notice a negligible performance differential for n-nucleotide lengths from 3 to 5. At these lengths, there is not enough

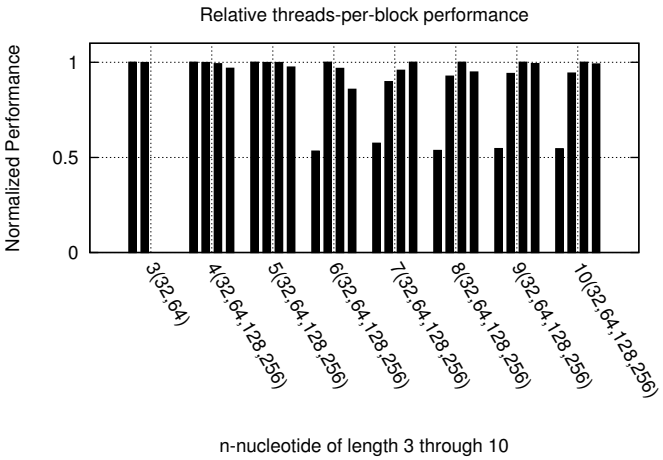

Figure 6. Normalized performance of each threads-per-block configuration (32, 64, 128 and 256) for n-nucleotide lengths from 3 to 10. For each n-nucleotide length, we take the threads-per-block configuration with the shortest execution time as the baseline. We then divide this value by the execution time of other configurations to generate the normalized performance. A value of 1 represents best performing configuration. A n-nucleotide sequence of length 3 can have only 32 or 64 threads-per-block.

workload to utilize all of the multiprocessors on the device, yielding little variation in the thread configurations. Once we reach an n-nucleotide length of 6 and greater, however, we notice there is nearly a 50% drop in performance if only 32 threads per block are used. This is due to the fact that we are limited by the hardware to having a maximum of 8 active thread-blocks per multi-processor. By only using 32 threads per block, we are limited to having a maximum of 8 active warps on a multiprocessor, with the maximum allowable being 48. Once we utilize above 32 threads, the performance significantly improves, since we are able to employ more active warps.

A. GPU Tuning and Findings

We show the thread configuration with respect to the n-nucleotide length in table III. Our kernel uses 33 registers per thread and limits us to 7 active thread-blocks per multiprocessor, which we reach when the length of the n-nucleotide addition is 8. As illustrated in figure 7, we achieve very little warp and thread block utilization on the GPU until we reach an n-nucleotide length of 8. However, at an n-nucleotide length of 8 and above, the utilization stabilizes at 87.5% for the thread-block utilization and at 58.3% for the warp utilization. We also examine the performance of our program using the number of instructions executed per second since the recombination problem does not require any floating point arithmetic. We achieve approximately 64% efficiency by attaining 430 billion instructions per second in our implementation, with the theoretical peak being 672 billion instructions per second on the GTX 480 GPU [21].

We define thread-block utilization as the ratio of the

Table III
THREADS PER KERNEL AND THREAD-BLOCK FOR N-NUCLEOTIDE LENGTHS 0 THROUGH 10.

GPU Thread and Thread-Block Configuration			
$\mathbf{N_{len}}$	$\mathbf{Threads_{Kernel}}$	$\mathbf{Threads_{Block}}$	$\mathbf{Num_{Blocks}}$
0	1	1	1
1	4	4	1
2	16	16	1
3	64	32	2
4	256	32	8
5	1,024	32	32
6	4,096	64	64
7	16,384	256	64
8	65,536	128	512
9	262,144	128	2,048
10	1,048,576	128	8,192

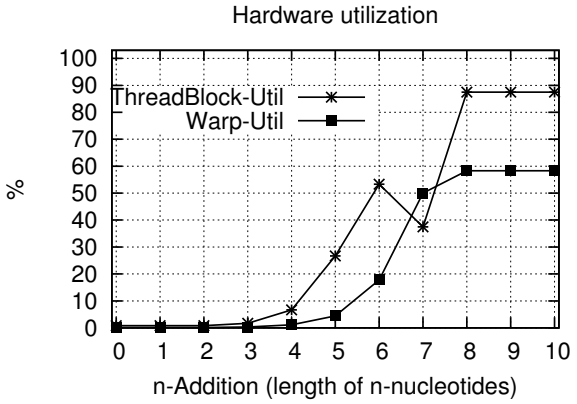

Figure 7. Device utilization in terms of active thread-blocks and warps.

number of active blocks we have on a given multi-processor over the maximum possible number of active thread-blocks. Similarly, the warp utilization is the ratio of the active number of warps we have over the theoretical maximum number of possible active warps. On the GPU, improving the warp-utilization further is possible by simply increasing the number of threads per block. However, this would lead to a reduction in the number of active blocks on the device and consequently, the thread-block utilization. For example, if 256 threads were used per thread-block at a n-nucleotide length of 8, then the thread-block utilization would reduce to 37.5%. We observe this trend in figure 6 specifically for n-nucleotide lengths of 8 through 10. If the register use could be reduced to 32, then we would be able to achieve 100% block utilization on the device.

For higher n-nucleotide lengths, since maximum thread-block utilization is reached, we try to maximize the active blocks and warps with the increased workload. During execution on the GPU, active blocks are swapped in and out of a multiprocessor during their lifetime. This helps hide the global memory latency by iterating through warps while waiting for memory requests. While it is important to have many active threads, it is suggested that at least 10 active warps per multiprocessor is sufficient to hide memory

Table IV
EXECUTION TIME, N-NUCLEOTIDE LENGTHS 0-10 ON GPU AND CPU.

In Silico Results		
$\mathbf{N_{len}}$	**GPU (min)**	$\mathbf{CPU(min)}$
0	31	9
1	38	15
2	46	44
3	57	197
4	66	999
5	77	4,843
6	95	17,434
7	323	49,193
8	1,031	NA
9	4,528	NA
10	17,170	NA

latency [21]. Once we reach a n-nucleotide length of 6, we have a total of 128 active warps and are able to hide most of the global memory latency. Since each thread is comparing its generated V(D)J sequences to a particular section of the *in vivo* set, each thread-block must have access to a space in the global memory equivalent to the largest subset of *in vivo* sequences. When n-nucleotide addition is of length 10, this size is relatively small at only 101.53 MB. This global memory usage is calculated based on the number of total thread-blocks. For example, if the n-nucleotide length is 10, then there will be (4^{10}) total threads in the kernel. If we use 128 threads per block, this means we will have 8192 thread-blocks. The global memory usage can then be calculated as *(number of thread-blocks)*(number of in-vivo sequences corresponding to given VJ pair)*(sizeof(int))*. Therefore, during the workload management, we mainly focus on maximizing active blocks.

The GPU timing results are shown in table IV for n-nucleotide lengths of 0 through 10. When we reach a n-nucleotide length of 8, we are at the maximum capacity in terms of active thread-blocks as illustrated in figure 7. Beyond this point, as the n-nucleotide length increases by 1, the execution time increases by around a factor of 4. This reflects the trend in the workload increase, which indicates that the amount of workload is heavily dominated by the 4^m possible nucleotide combinations. At a n-nucleotide length of 7, the GPU takes about 5 hours 23 minutes, while the single threaded CPU version takes approximately 34 days 3 hours and 53 minutes for the same task. Finding the number of different pathways to generate each *in vivo* sequence for up to a n-nucleotide length of 10 cumulatively takes 16 days 7 hours and 2 minutes on the GTX480 GPU. We estimate that the CPU version would take roughly 260 weeks to complete the same workload. Since the recombination process is highly data parallel, this workload could be distributed over approximately 110 Xeon cores operating at 2.83GHz to meet the GPU performance. Based on the execution time reported in table IV and the warp utilization reported in figure 7, performance of the algorithm could further be improved by

running the recombination process with nucleotide length (N) of 0 through 3 on the host machine using the CPU resources, and tailoring the GPU implementation to improve the warp utilization for nucleotide length of 4 through 6. However, execution time is mainly dominated by nucleotide length of 7 and above. Even if we could reduce the execution time to zero for smaller values of N, this would only lead to an overall improvement of less than 2%. If the execution time was reduced by half for the smaller values of N, we would see less than 1% overall improvement.

B. Biological Findings

From an immunological perspective, we set out to identify the validity of CRH's claim regarding the production (*in silico*) frequency of a certain sequence serving as a predictive index for its frequency *in vivo*. For the *in silico* frequencies to serve as a guide, these numbers need to have a positive correlation with the *in vivo* frequencies. Using the Shapiro-Wilk Test for normality, we initially determined that our dataset did not have a normal distribution ($p < 0.001$). Thus, to test for correlation, we applied the non-parametric Spearman's Rank Order Correlational Test on our dataset. The correlational coefficient was 0.056 ($p = 2x10^{-72}$), implying that the relationship between the *in silico* frequencies with the *in vivo* frequencies is minimal at best.

The plotted *in silico* ranks vs. the *in vivo* ranks are shown in figure 8. Many of the sequences have *in vivo* frequencies which map to a range of *in silico* frequencies, inconsistent with CRH's assertion that "*the relative frequencies with which sequences are produced are a good predictor of the spectrum of TCRβ sharing" [16].* Additionally, plotting the absolute *in silico* frequencies vs. *in vivo* frequencies (figure 9) does not support CRH's claim either, since the frequencies are inversely correlated. The genomics approach to sequencing large TCR datasets has been cost prohibitive to analyse a large number of mice. Even though our sample size is limited by 2 mice, our method provides a means of testing the veracity of the CRH on more comprehensive dataset.

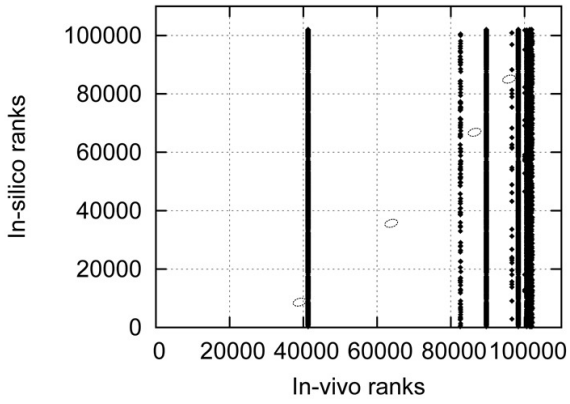

Figure 8. The Spearman Rank Order Correlational Test compares the *in vivo* sequence ranks with their *in silico* counterparts yielding a correlation coefficient of .056 ($p=2.499x10^{-72}$). The ellipsoid regions indicate the results predicted by CRH.

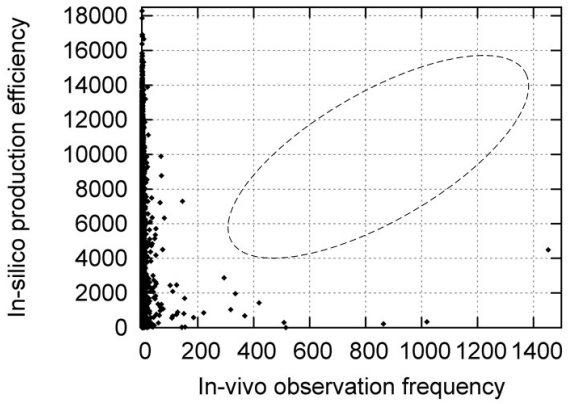

Figure 9. The absolute production frequencies at a n-nucleotide addition length of 10 (*in silico*) plotted against observation (*in vivo*) frequencies in the two mice. The ellipsoid region indicates the results predicted by CRH. Production efficiency represents the number of recombination pathways for each unique TCRβ sequence, whereas in-vivo frequency represents number of times each unique TCRβ sequence was observed in mouse dataset.

VI. Conclusion

Due to the extreme computational nature of V(D)J recombination, immunologists have employed approximation-based heuristic methods to study the immune systems of the jawed vertebrates. In this research, we have provided a way to model the generation of all possible TCRβ recombination paths with an affordable high performance computing platform. The original recombination algorithm presented earlier in [16], [17] has been completely reworked in order to effectively map the program architecture to the GPU hardware architecture. In addition to the threading utilization, we exploit several of the architectural features offered by the GTX 480 to achieve a high level of performance. These include coalesced memory transactions to the global memory, parallel reductions at a thread-block level in the shared memory, and efficient utilization of the constant memory and cache. We also provide a method for quickly creating a unique n-nucleotide sequence for each thread.

Based on our implementation, for the first time, we present a model of the mouse TCRβ repertoire on the order of $4x10^{14}$ recombination pathways. This is the first step towards conducting a peta-scale level of *in silico* TCRβ analysis by using a GPU platform to study the immune systems of complex organisms. While the GPU implementation of this code has demonstrated that this previously intractable biological problem is now feasible; further improvements utilizing techniques borrowed from Big Data (FM-indices),

high throughput signal processing (hardware based systolic arrays), or from combinatorial game theory (dynamic programming) could speed up this analysis and allow immunologists to attack larger data sets, including samples from large clinical cohorts. The recombination methodology presented in this paper can also be easily partitioned into a multi-GPU problem by partitioning n-nucleotide combinations into subsets among each GPU, and then further partitioning the n-nucleotide combinations to a one-to-one mapping per thread.

ACKNOWLEDGEMENTS

This work is supported in part by the iPlant Collaborative (funded by a grant from the National Science Foundation Plant Cyberinfrastructure Program, DBI-0735191), and NVIDIA CUDA Teaching Center program.

REFERENCES

[1] M. Gellert, "V(D)J Recombination: RAG Proteins, Repair Factors, and Regulation," *Annual Review of Biochemistry*, vol. 71, pp. 101-132, 2002, PMID: 12045092.

[2] D.G. Schatz and Y. Ji, "Recombination Centres and the Orchestration of V(D)J Recombination," *Nature Reviews Immunology*, vol. 11, no. 4, pp. 251-263, 2011, doi:10.1038/nri2941.

[3] J. Mansilla-Soto 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, 2003.

[4] S. Turner, N. Gruta, K. Kedzierska, P. Thomas, and P. Doherty, "Functional Implications of T Cell Receptor Diversity," *Current Opinion Immunology*, vol. 21, no. 3, pp. 286-290, 2009.

[5] F. Pandolfi, R. Cianci, F. Casciano, D. Pagliari, T. De Pasquale, R. Landolfi, G. Di Sante, J.T. Kurnick, and F. Ria, "Skewed T-cell Receptor Repertoire: More Than a Marker of Malignancy, A tool to Dissect the Immunopathology of Inflammatory Diseases," *Journal of Biological Regulators & Homeostatic Agents*, vol. 25, no. 2, pp.153-161, 2011, PMID: 21880203.

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

[7] M.R. Lieber, "Site-Specific Recombination in the Immune System," *The Journal of the Federation of American Societies for Experimental Biology*, vol. 5, pp. 2934-2944, 1991.

[8] A. Casrouge, E. Beaudoing, S. Dalle, C. Pannetier, J. Kanellopoulos, and P. Kourilsky, "Size Estimate of the Alpha Beta TCR Repertoire of Naive Mouse Splenocytes," *Journal of Immunology*, vol. 164, pp. 5782-5787, 2000, PMID: 10820256.

[9] G.H. Gauss and M.R. Lieber, Mechanistic Constraints on Diversity in Human V(D)J Recombination, *Molecular and Cellular Biology*, vol. 16, no. 1, pp. 258-269, 1996.

[10] V. Venturi, D.A. Price, D.C. Douek, and M.P. Davenport, "The Molecular Basis for Public T-cell Responses?," *Nature Reviews Immunology*, vol. 8, pp. 231-238, 2008, doi:10.1038/nri2260.

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

[12] J.J. Miles, A.M. Bulek, D.K. Cole, E. Gostick, A.J. Schauenburg, G. Dolton, V. Venturi, M.P. Davenport, M.P. Tan, S.R. Burrows, L. Wooldridge, D.A. Price, P.J. Rizkallah, and A.K. Sewell, "Genetic and structural basis for selection of a ubiquitous T cell receptor deployed in Epstein-Barr virus infection," *PLoS Pathogens*, vol. 6, no. 11, pp. e1001198, 2010.

[13] V. Venturi, M. Quigley, H. Greenaway, P. Ng, Z. Ende, T. McIntosh, T. Asher, J. Almeida, S. Levy, D. Price, M. Davenport, and D. Douek, "A Mechanism for TCR Sharing between T Cell Subsets and Individuals Revealed by Pyrosequencing," *Journal of Immunology*, vol. 186, no. 7, pp. 4285-4294, 2011.

[14] B.D. Rudd, V. Venturi, M.P. Davenport, 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, 2011.

[15] M. Yassai, D. Bosenko, M. Unruh, G. Zacharias, E. Reed, W. Demos, A. Ferrante, and J. Gorski, "Naive T cell repertoire skewing in HLA-A2 individuals by a specialized rearrangement mechanism results in public memory clonotypes," *Journal of Immunology*, vol. 186, no.5, pp. 2970-2977, 2011.

[16] 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," *Proc. National Academy of Sciences*, vol. 103, no. 49, pp. 18691-18696, 2006.

[17] 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," *Proc. of the National Academy of Sciences*, vol. 107, no. 45, pp. 19414-19419, 2010.

[18] Source code for the recombination algortihms and data, http://www2.engr.arizona.edu/~rcl/publications/source_codes/recombination/recombination.html, 2013.

[19] N. Fazilleau, J. Cabaniols, F. Lemaitre, I. Motta, P. Kourilsky, and J. Kanellopoulos, "V and V Public Repertoires are Highly Conserved in Terminal Deoxynucleotidyl Transferase-Deficient Mice," *J. of Immunology*, vol. 174, no. 1, pp. 345-355, 2005.

[20] M.P. Lefranc, V. Guidicelli, C. Ginestoux, J. Bodmer, W. Muller, R. Bontrop, M. Lemaitre, A. Malik, V. Barbie, and D. Chaume, "IMGT, Int. ImMunoGeneTics Database," *Nucleic Acids Research*, vol.27, no. 1, pp. 209-212, 1999.

[21] NVIDIA. NVIDIA CUDA compute unified device architecture programming guide 2013.

[22] H. Robins, S. Srivastava, P. Campregher, C. Turtle, J. Andriesen, S. Riddell, C. Carlson, and E. Warren, "Overlap and Effective Size of the Human CD8+ T Cell Receptor Repertoire," *Sci Transl Med*, vol. 2, no. 47, p. 47-64, 2010.

[23] H. Li, C. Ye, G. Ji, and J. Han, "Determinants of public T cell responses," *Cell Research*, vol. 22, no. 1, pp. 33-42, 2012.