Optimization of GPU-based Implementation of V(D)J Recombination Algorithm

Frederick Yen, Elnaz Tavakoli Yazdi, Ju Pan, Ankur Limaye and Ting-Chun Chiu
Department of Electrical & Computer Engineering, University of Arizona, Tucson, AZ 85721
Email: { fyen, tavakoliyazdi, pjokk722, ankurlimaye, tingcchiu }@email.arizona.edu

***Abstract*—In this study, we implemented two optimization techniques to improve the performance of the available baseline code. The first optimization was taking advantages of Open MPI to launch multiple GPUs for data operations. Even though the evaluated performance tests were only conducted on MPI-based code for two GPUs, with appropriate optimization like modification of reduction algorithm and reallocation of thread and block size, it demonstrated at least a 2× speedup. The second optimization method was performing bit-wise operations in the recombination process. The input data sets was converted to the binary domain with four characters packed into one byte using encoding technique. This encoding scheme resulted in the constant memory footprint by 3.5× and global memory footprint by 4×. The kernel was modified to operate and compare the new data sets. Even though, the second optimization significantly reduces the memory footprint, there is no significant performance improvement in terms of execution time.**

***Keywords*—GPU, CUDA, OpenMPI**

I. Introduction

The adaptive immune system is one of the immunity strategies found in the jawed vertebrates. This system relies on generating diverse repertoire of antigen receptors (immunoglobulins (Igs) and T-cell receptors (TCRs) [1]. The mechanism responsible for creating this huge repertoire is a specialized DNA recombination process – the V(D)J recombination [2]. The TCR is composed of an α and β chain (TCRα and TCRβ). Both of these chains undergo the recombination process. It consists of rearrangement of variable (V), diversity (D), and joining (J) segments chosen from members of each gene family [2]. The V(D)J recombination is highly sophisticated mechanism and consists of several sub-processes like the palindromic extension, chew back, and concatenation on both DJ and VD sides. These combinations of the V, D, and J gene segments make it possible to construct unique receptors. The possible $\alpha\beta$TCR repertoire size in mice is estimated to exceed 10^{15}, and more than 5×10^{12} unique TCRβ) chains [3].

The immunologists find it difficult to understand the TCRβ repertoire and the underlying proclivities due to its huge size. Currently, the immunologists model the TCRβ repertoire in mice based on the analysis of minor subset of the total TCRβ repertoire followed by extrapolation to the global repertoire. However, since each mouse represents a small sampling (10^{-7}) of the potential TCRβ repertoire, even the extrapolation method has it's limits. It is hard to predict the outcomes of immune responses to vaccines, unless all the 5×10^{12} unique TCR sequences are modeled (*in silico*) and compared with the number of times each sequence is generated by different recombination paths.

In [4], the researchers used GPU to model the TCRβ repertoire, and were successful in generating TCRβ sequences in the order of 4×10^{14} recombination pathways (1×10^{8} fold higher than prior studies). In our project, we intended to optimize the work in [4], to make the code scalable and memory-efficient. We performed three key optimizations during our project.

1) Implemented the baseline code of [4] on Kepler K20Xm GPU, and resolved the memory exhaustion issue encountered.
2) Improved the scalability of the baseline code by executing the code on multiple GPU nodes.
3) Reduced the global-, shared- and local-memory usage by encoding the data in 2-bit representation instead of 8-bit used in [4].

The project report is organized in the following way: The existing implementations and their drawbacks are discussed in Section II. A detailed explanation regarding our optimization strategies (multi-GPU implementation and bit-wise representation), and the experimental results are presented in Sections III, IV and V.

II. Related Work

The first attempt to model T-Cell Receptor β (TCRβ) sequences from two mice [5], [6] was written in Python and was executed on $2.83GHz$ quad-core Intel Xeon processor having $2GB$ RAM/core. It was a sequential implementation, and had two major shortcomings: It could model the TCRβ sequences for n-nucleotides only till length 7. It could not model the entire TCRβ repertoire. Moreover, the random sampling method was utilized to find possible sequences. This resulted in a biased selection and decrease in the performance. Even with fewer nucleotides support, it took more than 34 days to generate the sequences.

The first work that successfully modeled all the unique possible TCRβ sequences [4] was a CUDA implementation and used GTX480 GPU for executing the code. The authors restructured the algorithm and made it highly parallel executable to generate the TCRβ sequences with maximum length of 10 for the n-nucleotides. The authors carefully exploited the memory hierarchy, partitioned the data evenly and allocated it on registers, caches, shared and global memory to deal with the high amount of data. They even eliminated the communication among GPU threads by generating a unique n-nucleotides by each thread. Even after performing such optimizations, the GPU-based code required around 16 days to find all the V(D)J

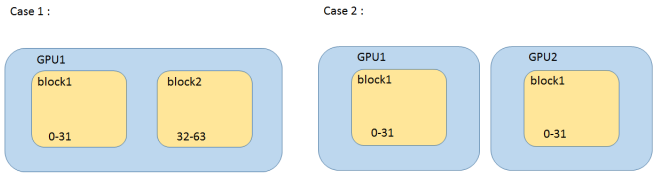

Fig. 1: Wrong mapping strategy – Case 1: 1 GPU, and Case 2: 2 GPUs, with id = threadIdx.x + blockDim.x $\times$ blockIdx.x

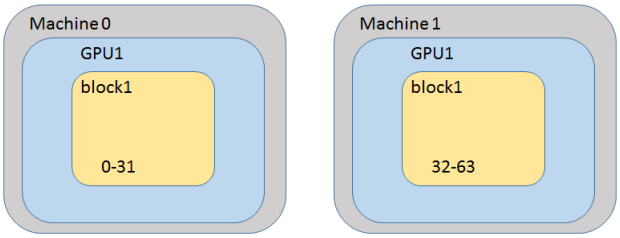

Fig. 2: Using Machine index and Machine dimension into index declaration. Index = threadIdx.x + blockIdx.x $\times$ blockDim.x + machineIdx.x $\times$ machineDim.x

recombination paths for n-nucleotides with length from 0 to 10.

III. Methodology

We used the Tesla K20Xm GPU provided by the ElGato system as the execution hardware. The Tesla K20Xm GPU offers advanced features compared to the GTX480 GPU used by [4], and reduces the execution time significantly. The most notable feature of K20Xm is that it has 14 multiprocessors that have more active blocks availability compared to GTX480. Moreover, the Tesla K20Xm GPU has larger shared memory size per block than GTX480. The ElGato system also provides for multiple GPU usage, which was useful for our multiple-GPU implementation. As the hardware we used for our implementation is significantly better than the one used by [4], we first implemented the baseline code on K20Xm without any optimizations, and used this results for a fair comparison.

The optimizations performed are discussed in details in the following sections: Section IV describes the multi-GPU implementation using MPI, while the bit-representation and bit-wise operations modifications are described in Section V.

IV. Multi-GPU Implementation using MPI

Before performing any optimizations, we ran the baseline code on ElGato system and encountered two issues: Memory exhaustion and System run-time limit. After performing analysis, we discovered that the main reason causing memory exhaustion was that the architecture of the K20Xm is vastly different from the GTX480. In the baseline code, each block uses 576×4 bytes of shared memory for n-nucleotide comparisons. In the GTX480, there are only 3 active blocks and hence the shared memory allocated is 6912 Bytes (576×4 Bytes $\times 3$ blocks). However, when the K20Xm GPU in ElGato is used, a maximum of 56 active blocks are created. This makes the total needed shared memory be 126 KB, which cannot be provided by the architecture and this causes memory exhaustion. The second issue is not related to the hardware but the limit of queue time of the ElGato system. The ElGato system kills the job if it exceeds the run-time limit. Currently, the run-time limit is two days. The execution time for larger n-nucleotides exceed two days, and hence we are not able to complete the execution. In order to overcome the mentioned issues, we focused our optimization tasks on workload separation among GPUs and modification of the algorithm structure in some parts.

A. CUDA-aware OpenMPI

For every parallel processing technique, programmers should cautiously pay attention to partitioning the tasks on different machines and reducing results from different machines to get the correct answer. When we scrutinized the baseline code, we realized that each thread creates a unique n-nucleotide based on their thread index, and the total number of threads would be 4^n. Intuitively, distributing 4^n threads evenly to different GPUs should be a fair approach. However, it would cause a serious error if we separate the workloads without further detailed setups. For instance, if 64 threads were allocated to two GPUs, each of them will launch threads with indexes from 0 $\sim$ 31 and perform the same tasks. This is illustrated in Fig. 1. To avoid this problem, we introduced the concepts of machine dimension and machine index. We took advantage of the fact that each machine knew its rank in MPI environment and used it while calculating the index value in the kernel function. Fig. 2 illustrates this concept. The machineIdx is the rank of machine in MPI environment and machineDim is (blockDim $\times$ gridDim). This ensures that distinct machines do not launch overlapped indexes and the performance corrects functionality.

After performing the multi-GPU implementation process, a reduction of the output is needed to obtain the correct answer as each machine has it's own array of computation results. We used the MPI_Reduce function given by Open MPI for the reduction operation. This function sums all the result in h_Result in the root machine (the machine with rank = 0) and print the final result. The reason behind choosing one machine to print the final result is that each machine only has partial result of the final output.

B. Algorithm Optimization and Configuration Strategy

In the baseline code, the kernel function is divided into two parts: comparison and reduction. The comparison part performs very well and there is no need for further optimization. However, the reduction function used by the author is not an optimal solution and can seriously slow down the computation process (see Figure 3). The size of Result_sm array is set equal the maximum block size to ensure that each thread can store the partial sum in Result_sm array corresponds to its index for all conditions. After the thread synchronization, the kernel function needs to perform the reduction operation for nine times to store the correct sum in Result_sm[0]. To speed up the process, we substituted the reduction algorithm with atomicAdd function and reduced the the size of Result_sm to 1. In Fig. 3, we have shown the speedup of our optimization. Since

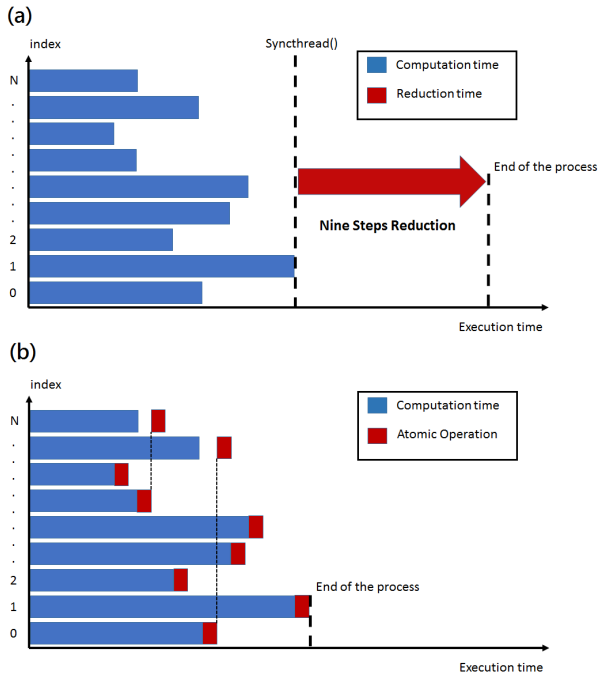

Fig. 3: Visualize the runtime of (a) Reduction Algorithm and (b) Atomic Operation.

each thread has a different execution time for comparisons, we are able to hide the run-time of atomicAdd into the comparison part and reduce the total computation time. Additionally, this approach also prevented shared memory exhaustion since each block only needs (65×4 B) shared memory. The total required shared memory size is 14.22 KB when the kernel launches maximum of 56 blocks in the GPU.

Considering the grid and thread block configurations of the GPU, utilizing the computing power of the GPU as much as possible was one of our goals. This implied that we aimed for high occupancy for our chosen configuration. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at the same time. Having a higher occupancy does not lead to higher performance in all situations, but it is an essential metric for measuring the latency hiding ability of the kernel.

We used the CUDA Occupancy calculator provided by NVIDIA to ensure that our chosen configuration achieves high occupancy. The Table I shows that for $n \leq 4$, the GPU needed to launch only one block and the occupancy is fairly low. This implies that even though 256 threads are sufficient for the calculations, the GPU has not fully take advantage of its computing components. But for cases where $n \geq 6$, the occupancy is 100% for all, implying that the total number of threads needed are large enough for the GPU to launch maximum number of active warps supported by the hardware at the same time, and hence utilizing all the GPU's computing power. As a result, we conclude that we have implemented the best thread block configuration that fully takes advantage of the GPU.

TABLE I: Occupancy table for n = 4 $\sim$ 7

n	Total threads	Threads per block	Blocks	Occupancy
4	256	256	1	12.5 %
5	1,024	256	4	50 %
6	4,096	256	16	100 %
7	16,384	512	32	100 %

N length	0	1	2	3	4	5	6	7	8	9	10	11	Total time (Days)
GTX 480	31	38	46	57	66	77	95	323	1031	4528	17170	N/A	16.3
Single-K20xm without any optimization	31	65	78	90	95	107	Memory Exhaustion	Memory Exhaustion	Memory Exhaustion	Memory Exhaustion	Memory Exhaustion	N/A	N/A
Single-K20Xm with optimization	30.5	35	41.2	48	48	54.8	106	87	418	838	Reach the runtime limit (55%)	N/A	>3.18
Multi-K20Xm with optimization (2GPU)	30.5	35.2	41.3	48	47.8	53.9	61.15	72.48	191.92	551.78	2237.8	Reach the runtime limit (30 %)	2.34

Fig. 4: The experimental result of CUDA-aware OpenMPI implementation.

C. Result and Validation

The Fig. 4 presents our experiment results of multi-GPU implementation with the run-times for various n-nucleotide lengths are shown in minutes (except the final column). The first row is the result reported in [4], while the other results are for our implementations on the ElGato system. It can be seen from rows 1 & 2 that without any optimization, the baseline code performs worse on the K20Xm with increased execution time and the shared memory exhaustion problem. There are two possible reasons for poor performance: The first reason is that since K20Xm allows for launchomg at most 14 times more streaming multiprocessors than GTX480, the reading and writing in global memory can slow down the execution. The second reason is due to poor performance of the reduction operation. It was discussed in Section IV-B. After optimizing the reduction operation, there is about $2\times$ speedup (for n = 1 to 5), and there is no shared memory exhaustion. However, the performance of single GPU can still be optimized since it reaches the run-time limit for n = 10. When n = 10, it completed only 55% of the process, and took 2880 minutes (48 hours). To overcome these, we implemented the multi-GPU version and ran it on two GPUs. It is evident from Fig. 4, the MPI-based program is performs better when the n is larger than 5. This happens because the GPUs can launch more active threads in the computational phase and achieve speedup.

We validated our results by comparing our results with the results from [4]. We wrote a python script to compare the results. It uses the diff command in terminal across all the result files to check for differences between two files. The comparing outcome shows that all the results from our MPI-based code are exactly the same as in [4]. This confirms that our implementation for using Open MPI is successful and efficient. Our MPI-based implementation shows an overall speedup on two GPUs, and has at most $8.2\times$ speedup (for n = 9), with average speedup of $6.8\times$, compared to the baseline code implementation on GTX480.

TABLE II: 2-bit Encoding scheme

Gene base	2-bit representation
A	00
T	01
G	10
C	11

V. Bit-wise Representation

The second optimization direction was reducing the memory footprint of the implementation. To achieve reduction in memory sizes, we decided to encode the four bases – A, T, C and G using 2-bit representation instead of 8-bit representation used in [4]. We describe our two phases used for implementing bit-wise version of the available baseline code in this section. The first phase consisted of converting input data sets from char (8-bit) domain to binary (2-bit) domain. In the second phase we modified the baseline algorithm to implement bit-wise operations on the new input data sets. The following subsections describe the implementation of these two phases in details.

A. Conversion of Input data sets to 2-bit representation

The input data set contains all the possible V, D, and J genes and *in vivo* sequences. There are 20 basic V genes, 2 D genes, 12 basic J genes and 101837 *in vivo* sequences. Each of V and J genes are stored in separate text files consisting the sequence length and the corresponding gene sequence in each line of the file. The format of D genes is a bit different, having sequence length, gene sequence and the number of times that the gene sequence can be generated in each line. There is only one text file containing all the *in vivo* sequences, consisting of the V gene index, J gene index, sequence length and the corresponding *in vivo* sequence per line of the text file.

Based on the baseline code, the V, D and J genes data need to be stored in the constant memory. Therefore, it is necessary to store the input data set in 2-bit representation format in the constant memory. This conversion is handled by the CPU, and performed in 'TNT_gold.cpp' file. The encoding scheme is shown in Table II. The sequences are read from the input files, 4 characters at a time, and encoded using the encoding scheme, to get the encoded 8-bit (char) value. If there are less than 4 characters left in the line, '00' bits are padded to maintain the 8-bit encoded values. This encoding scheme results in compact representation of the V, D and J gene sequences, and hence reduces the constant memory footprint of the code. The *in vivo* sequence is encoded in the similar fashion with one difference. Each line of the *in vivo* is encoded to a fixed 16 B width. This is to ensure that the starting address of each *in vivo* sequence is a multiple of 16. The *in vivo* data is stored in the global memory, and the threads in a block bring one sequence at a time to shared memory.

We validated the encoding scheme, by decoding the data and comparing it with the original data.

B. Kernel modification

We needed to modify the baseline kernel to perform combination process on new input data sets. The baseline kernel in [4] can easily perform recombination process and comparison, as each character of data is stored in one byte in the global or constant memory. However, in our optimization, we encoded multiple gene bases in a single byte. In this section, we explain our new algorithm to perform bit-wise operations on new data sets.

In the baseline code, the GPU starts its execution by determining how many and which *in vivo* sequences will be compared based on different V and J sets. 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 have to ensure that there are no duplicate n-sequences. To accomplish this, we took advantage of available function in the baseline code. However, since the data is encoded, we had to encode the n-sequence using same technique. It should be mentioned that we defined local register instead of array since the maximum length of n-sequence is 3 Bytes instead of 12 Bytes required in [4]. In addition, the new function assigns a unique n-sequence combination to each thread given the length of n-sequence, a thread ID, a block ID, and thread-block dimensions. We start the recombination process after each thread determines its own n-nucleotide sequence.

There are four main loops in the GPU kernel. The first for loop iterates through all the *in vivo* sequences. The *in vivo* sequence is loaded from global memory and is stored into the shared memory. To accomplish this, each thread brings one byte of *in vivo* sequence to the shared memory. However, each thread needs to bring more than one byte of sequence to the shared memory if the block dimension was less than 16 which is the maximum length of *in vivo* sequence. We need to be aware of length of current sequence to use it in further loops. To extract sequence length, we need to perform mask operation on second byte of *in vivo* sequence since it encoded in this way. It should be mentioned that we assign five bits to index of V gene since the maximum number for index is 20. We store index of V gene in the first byte of *in vivo* sequence. In addition, we devote five bits to the index of J gene. However, the first two bits of this index is stored in the first byte of *in vivo* sequence and the last three bits is stored in the second byte of *in vivo* sequence. The maximum length of *in vivo* sequence is equal to 54. Therefore, we devote six bits to sequence length which is stored in the last six bits of second byte.

The second for loop iterate 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. We need to consider two steps for comparing V with the beginning of the *in vivo* sequences. Based on the length of current V sequence, we define a for loop which iterates through full bytes of V sequence. Indeed, we compare the full byte of V sequence with full byte of *in vivo* sequence in the first steps. In the second step, we compare the remaining characters of V sequence with the corresponding character of *in vivo* sequence. If there is a single mismatch between V and $invivo$ sequence, we will load a new *in vivo* sequence without combining D, n and J sequence to that particular V sequence. It should notice that we need to keep track of current

byte of $invivo$ sequence and overflow bits of each loop as we proceed. 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. As previous loop, we terminate the current comparison, if there is a single character mismatch. However, we will load new D sequence if there is a character mismatch. We need to follow new method in this loop since the n-sequence characters can be placed on either side of the D sequence. To accomplish this, we define one inner for loop which iterates through all possible combination of nDn' sequence. Based on the position of for loop, we start comparing full bytes of n-nucleotide sequence with *in vivo* sequence in the first step. Subsequently, we compare overflow bits of n sequence with corresponding part of *in vivo* in the second step. However, If there is a single character mismatch we will proceed to the next iteration of current for loop which reduce the length of n part and increasing the length of n' part. If the n-nucleotide does match with its corresponding part of *in vivo*, we will proceed to compare D sequence. We utilize previous two steps to compare D sequence with *in vivo* sequence. If the nD sequence does match then we proceed to combine n' sequence to nD sequence. Again, we use the same strategy to compare n' sequence. It should mention that we need to record the current byte of $invivo$ sequence and overflow bits of each loop as we proceed. It should be noted that we compare each of these generated sequences one byte at a time from the last byte 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 byte at a time to the latter portion of an *in vivo* sequence. We use the same method as first loop to completely compare V sequence with *in vivo* sequence. If there is a character mismatch, we load new J sequence to perform comparison. 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.

C. Results

The improvements in the memory requirements is tabulated in Table III. The variables DB1, V, and J are stored in constant memory and has average 70.99% reduction in memory requirements. The *in vivo* is stored in the global memory and has reduction of 75% requirements.

TABLE III: Memory Requirements

Variable	Old Memory Size	New Memory Size	% Decrease
DB1	1448 B	425 B	70.65 %
V	3107 B	913 B	70.61 %
J	3210 B	908 B	71.71 %
in vivo	6517568 B	1629392 B	75.00 %

VI. Conclusion

In this project, we evaluated the performance improvement of two different optimization techniques on the available baseline code. It should be noted that both implementations show the outstanding result in comparison with the baseline code on GTX480. In the first optimization method, we took advantage of using multiple GPU. We modified the reduction algorithm and adjusted the block dimension, grid dimension and the constant memory size to avoid the shared memory exhaustion. As a consequence of using multiple GPU, the number of active threads increased, and resulted in $6.8\times$ times improvement in execution time in comparison to the available baseline code. In the second step of optimization, we utilized bit-wise operations to implement recombination process. As a result of this technique, we achieve a significant reduction in term of memory requirement (constant memory requirement by $3.5\times$ and global memory requirement by $4\times$). However, we did not observe significant performance improvement in term of execution time since threads' workload did not decrease substantially. In addition, we have more complex kernel since we are not dealing with one character as an one byte. We think that combining two techniques will result in significant improvement in term of both memory usage and execution time. The new combined method can take advantage of low memory usage of bit-wise version as well as increasing the number of active threads due to using multiple GPU. Therefore, we can utilize all the mentioned improvements to map different time-consuming processes on to the GPU and achieve desirable performance.

References

[1] M. Gellert, "V(D)J Recombination: RAG proteins, repair factors, and regulation," *Annual Review of Biochemistry*, vol. 71, no. 1, pp. 101–132, 2002.

[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.

[3] M. R. Lieber, "Site-specific recombination in the immune system," *The Journal of the Federation of American Societies for Experimental Biology*, vol. 5, no. 14, pp. 2934–2944, 1991.

[4] G. Striemer, H. Krovi, A. Akoglu, B. Vincent, B. Hopson, J. Frelinger, and A. Buntzman, "Overcoming the limitations posed by tcr-beta repertoire modeling through a gpu-based in-silico dna recombination algorithm," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 231–240.

[5] 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. 18 691–18 696, 2006.

[6] 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 naive T-cell repertoire," *Proc. National Academy of Sciences*, vol. 107, no. 45, pp. 19 414–19 419, 2010.