# *Case Study for DNA Codons Occurrences*

Authors:
Carlos Sanchez
Darilys Pereira

*Abstract*:

This program seeks to parse a file containing DNA information, namely a large codon sequence and proceeds to register the occurrences of each codon. After completion, the function must return a list containing an overview of the total presence of codons in the sequence parsed; this information can be used by biologists to perform different kinds of statistical analysis. The task at its core seems trivial, as the process of reading and matching strings from a text file is not inherently complex. The main difficulty is that the cost of doing so using the computer's CPU is very high. This is not immediately evident when processing a small file, however when dealing with the usual larger files that are used to describe DNA sequences the cost of processing becomes clear. The approach chosen for this program was to use the NVIDIA graphics card GPU to try to parallelize this problem, and using parallelism instead of reading and processing the DNA file in a linear fashion.

## *Introduction*:

The genetic code is the information contained in DNA sequences, living cells use this information translating it into proteins. DNA strands are composed of simple units called nucleotides, and each nucleotide consists of one of four nucleobases cytosine(C), guanine(G), adenine(A), or Thymine(T). In general, the genetic code is composed of "words" that contain 3 nucleotides which are named codons (e.g.: ACT, ATC, there are about 64 different codons which is the max number of groups of 3 letters that can be formed using one out of 4 letters). Any regular genetic sequence is made of billions of combinations of these codons. Our program allows to scan through a file containing a DNA sequence and compute the occurrences of every codon present presenting this information to the users after completion. In order to solve this problem, the program will make use of an NVIDIA graphics card and implement a parallel solution which will ensure that the file is read in a parallel way, meaning that multiple threads will work on the file simultaneously, ensuring that the program runs in significantly less time.

## *Related Work:*

There exists an extensive amount of work closely related to the problem being addressed, one of the most important is the "Huffman Codes", which is a family of algorithms that are used for lossless data compression [6]. The Huffman Codes provide a set of algorithms and solutions which are used to represent DNA sequences in memory by encrypting them using the least amount of bits possible to represent each nucleobase. The most basic implementation of the Huffman Algorithm can be considered as a two pass algorithm that navigates through the file and computes the occurrences of each nucleobase, and creates a bit representation for each depending on their rate of occurrence. This algorithm is brilliant as it allows to tremendously reduce the amount of memory needed to store very long DNA sequences. At the time when the "Huffman Codes" was published computers didn't count with the amount of RAM memory that modern computers have today, and memory management was given top priority. However, at present time, modern computers have large amounts of RAM at their disposal, it is based in this premise that our program's approach to this problem diverges, and a bigger emphasis is placed on execution time and memory management is pushed further back in the stack of priorities.

## *Process:*

The thought process followed when computing the solutions provided is described in full in the following section, however the following definitions allow to comprehend the text more easily:

**Codon:** A group of 3 letters, where each one of them is A, C, T or G.

**Codon polynomial value:** This is the value assigned to a codon by evaluating it using the following formula:

$$letter\_value1 * (base)^{pos} + letter\_value2 * (base)^{pos} + letter\_value3 * (base)^{pos}$$

where each letter value is assigned statically at runtime as is the base which is 4, the *pos* is the position of the letter in the codon, e.g. in a codon such as: "ABC" the letter A is evaluated as the value of A times the base to the position of ‘A’ which is zero.()

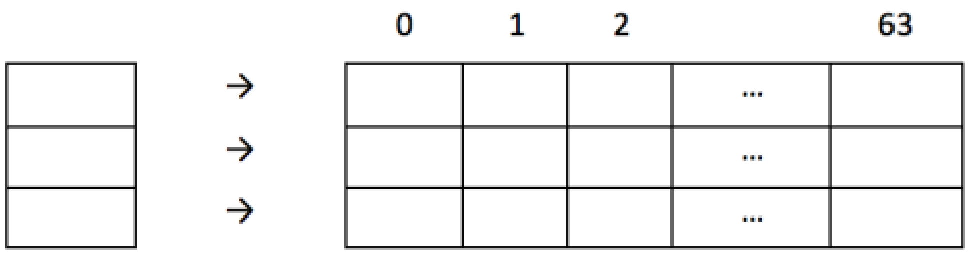
**Thread ID**: This is the unique identifier that allows to differentiate a thread from another within a core in the GPU.

**Overall Thread ID:** This is the global thread id that is computed by considering the current thread's position using its block id and its block dimensions.

Before arriving to the final solution, a few experiments were made before the right configuration of data structures, number of cores and threads could be determined based on the file sizes that were used. During all of the written solutions the program would read a file from memory and load it into the local RAM, then the contents of this memory would be copied into the GPU’s local memory in preparation for parsing.

On the first attempt the program would continue to create a two dimensional array in the GPU, the number of rows on this array was defined by the number of codons in the given file and the number of columns was determined by the number of possible codons, which is 64.

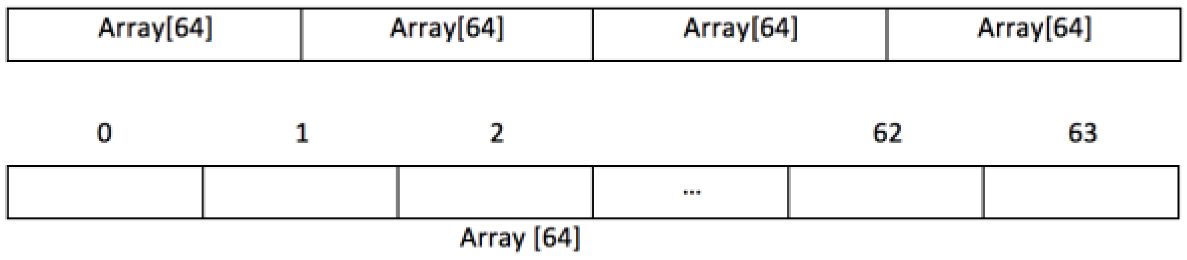
**Fig 1.1 Two Dimensional Array. (Array of pointers)**



This table was used in conjunction with multiple threads to work through the whole file simultaneously, each thread would access a unique codon in the file and evaluate it as a polynomial using a base of four and each of the 3 letters position to assign each codon a unique index from 0 to 64, the thread then continues to access a unique row in the previously created table using its overall thread id and writes to the column that matches the polynomial value of the string that was evaluated. After the completion of all threads, the program would perform a parallel vector addition of the results on each one of the column threads, for this task 64 threads would be used in parallel to compute these calculations and store the results into a corresponding column in a results array of integer numbers. This result array would be presented to the user upon completion before exiting the program. While this approach provided a solution, it didn’t work as intended for larger files as the amount of memory used was extreme, for example a file of 100 codons or 300 characters would require a row in the threads table for each codon, therefore for every codon an integer array of 64 elements was pre-allocated and filled beforehand, 100 codons would require an additional memory of 100 * 64 * 4 = 25,600 bytes to compute. This approach proved to be slow for moderately sized files (< 1 or 2 Mb) and provided erroneous results when dealing with larger files.

During the second attempt the first approach was attempted again; this time the two dimensional array was substituted by a one-dimensional array composed of arrays of 64 integer numbers each. The purpose of this was to circumvent some of the limitations of the previous usage of a 2D array, each thread was assigned a codon and every array was used to store the results of one thread. After the file was parsed and each thread recorded the occurrence of its assigned codon, the total amount of occurrences would be computed by using 64 threads just as before and visiting each sub array at the current thread's position.

**Fig 1.2 New Structure Used**



This approach proved to be an improvement over the original idea as the memory allocation of a one large one dimensional array was less demanding on the GPU kernel and the program executed without errors, the solution still used large amounts of memory and proved to be slow when parsing large files. The computation times while better than the original 2D array approach were far behind the linear solution.

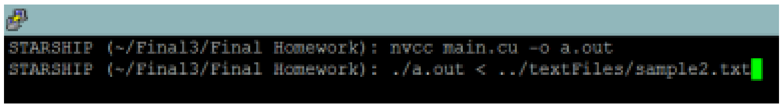
It was clear after this iteration that the amount of memory being used by the program was a largely contributing factor in the runtime complexity of this program.

## The final solution:

The final solution which is the one included in this paper is the result of the experiences learned from the previous iterations of the program. It was clear at this point that memory was an important factor and that each thread was not doing enough work. Until this point, each thread was being used to compute only one codon from the sequence, however as discussed before this approach was inherently bad as the amount of memory needed for this would be directly proportional to the number of codons in the file. This time a decision was made to allow each thread to compute multiple codons, after some experimentation it was decided after multiple trials that to parse a file of approximately 200MB of size the best number of codons assigned per thread would be 64, that way each thread would work on a substantial amount of codons and the memory usage would be reduced, compared to the previous calculations the memory requirements is reduced by a factor of 64! These new implementations alone allowed the program to parse a 200MB file faster than the linear version of the algorithm, the speed reduction was roughly a factor of five.

The program is a command line application that relies in the use of Pipes in UNIX to provide a file through standard input (*stdin*), a simple program call is done as follows:

**Fig 1.3 Command line**

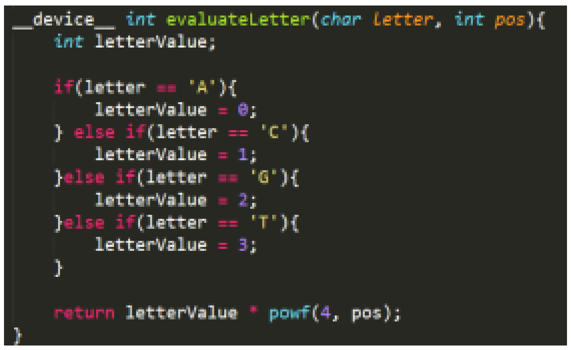


```
STARSHIP (~/Final3/Final Homework): nvcc main.cu -o a.out
STARSHIP (~/Final3/Final Homework): ./a.out < ../textFiles/sample2.txt
```

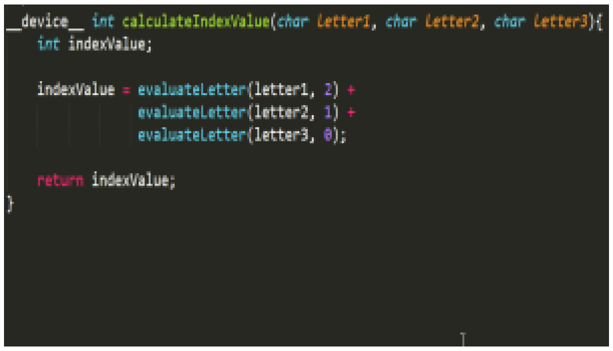
The program then takes the file and loads its contents to RAM, allocates the needed memory on the GPU for the contents of the file and copies it. After the file is loaded to the GPU the program proceeds to allocate the needed "thread list" which is the linear array where each thread will record their work. The file where the final results will be recorded is also created here.

At this point the program goes through the contents of the file in cuda and assigns each thread a portion of the file, in this case each thread is assigned 64 codons and each thread evaluates each of their assigned codons and continues to record their findings into their assigned array. Each codon is evaluated by determining its polynomial value.

**Fig 1.4 Determining Codon's Value**



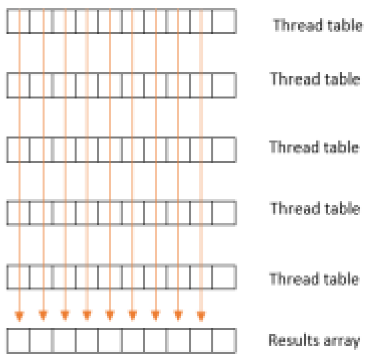
```
__device__ int evaluateLetter(char letter, int pos){
    int letterValue;

    if(letter == 'A'){
        letterValue = 0;
    } else if(letter == 'C'){
        letterValue = 1;
    }else if(letter == 'G'){
        letterValue = 2;
    }else if(letter == 'T'){
        letterValue = 3;
    }

    return letterValue * powf(4, pos);
}
```

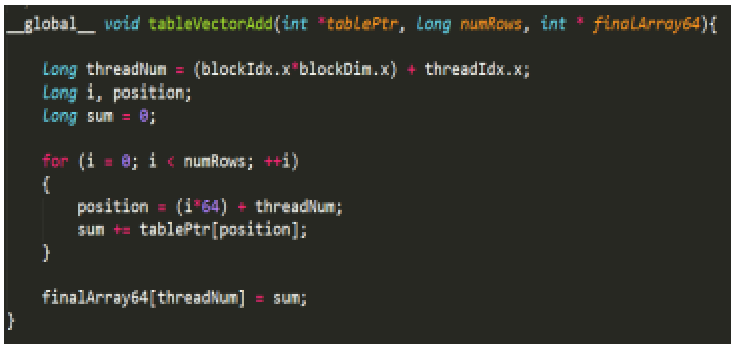


```
__device__ int calculateIndexValue(char letter1, char letter2, char letter3){
    int indexValue;

    indexValue = evaluateLetter(letter1, 2) +
                 evaluateLetter(letter2, 1) +
                 evaluateLetter(letter3, 0);

    return indexValue;
}
```

The second step of the algorithm uses parallelism to compute the vector addition of all of the integer arrays of size 64 on the thread table, the thread table in this case will have a number of rows that is determined by the number of codons in the file divided by 64. In order to perform this step, 64 threads are used to compute the vector addition of every entry in a specific index of each sub array, this index is determined by the thread number, ranging from zero to 63. Each thread then computes their assigned column's vector addition and records the results into a previously defined integer array of 64 using its thread ID to determine the column to modify.

Fig 1.5 Vector Addition





```
__global__ void tableVectorAdd(int *tablePtr, Long numRows, int * finalArray64){

    Long threadNum = (blockIdx.x*blockDim.x) + threadIdx.x;
    Long i, position;
    Long sum = 0;

    for (i = 0; i < numRows; ++i)
    {
        position = (i*64) + threadNum;
        sum += tablePtr[position];
    }

    finalArray64[threadNum] = sum;
}
```

After this step is completed, the resulting array contains the results of all of the 64 threads vector additions results for their respective columns, this array is indicative of the statistical information of the file that was parsed and processed and copies the result array from the GPU into the CPU so that further data analysis can be performed if needed.

## ***Results***:

At the completion of the final program, the benefits of parallelism were clear, and became more explicit as the file being parsed increased in size. During testing, a  file of size 200 MB was parsed through in about 1210 ms using the parallel algorithm, while the linear version took 6043 ms to complete, using  a larger file of about 600MB, the program took about 2536 ms to complete,  beating the linear version of the program which took some long 17265 ms to complete. As it is shown in the following table the benefits of parallelism were clear and undisputable at this point.

**Table 1.6 Execution times vs File Size.**

| File Size | Parallel Function Execution Time | Linear Function Execution Time |
|---|---|---|
| 100KB | 163 ms | 0.009 ms |
| 200MB | 1,210 ms | 6,043 ms |
| 600MB | 2,536 ms | 17,265 ms |

Although the program uses considerable memory to parse through the DNA text files, the improvements on execution times of the parallel version of the function  clearly outrank the linear version of the function, as shown on **table 1.6**. Notice that to fully enjoy the perks of parallelism files must be of reasonable size, as the cost of copying the data from the computer into the GPU is considerable. This can be clearly appreciated in the previous graph, where the performance gain in execution time is negligible and even negative for smaller files where a linear solution proves to be considerably more efficient. However, this is not significant as DNA files are extremely large and will always benefit from parallelism.

## ***Conclusions*:**

Upon the completion of this project, it is clear to conclude that this problem clearly benefits from parallelism, the performance gains on execution time grew as the file size increased. The only immediate limitation on this approach is the limited amount of threads and cores at the developer's disposal in the GPU, this would render impossible to parse extremely large files with the current configurations as there are not enough threads available to process a very big file in full. Further immediate enhancements to this program should allow to divide any file into chunks of considerable size as needed, these files will preferably be of the maximum size that the current GPU limitations allow to parse at a time, this would hypothetically allow to process files of any size successfully while still benefiting from the advantages of parallelism. Other modifications that would further augment the user’s experience would be to add more functionality to the final stage of the function in which the results are presented to the user, one modification the creators of this program propose would be to provide user’s with the frequency of occurrence of each nucleotide or letter which would further contribute to the statistics provided by the program.

## References:

1.Accelerated Computing. (2016). Retrieved December 02, 2016, from https://devtalk.nvidia.com/default/topic/509940/what-is-the-limit-of-cudamalloc-/

2."CUDA Toolkit Documentation V8.0." CUDA Toolkit Documentation. N.p., n.d. Web. 01 Dec. 2016, from http://docs.nvidia.com/cuda/

3.Cickovski, Trevor. Computed Unified Device Architecture(CUDA) Tutorial. N.p.: n.p., n.d. 3 Aug. 2016. Web. 17 Nov. 2016, from https://moodle.cis.fiu.edu/v3.1/pluginfile.php/87884/mod_resource/content/0/CUDA_Tutorial.pdf

4."UCSB Science Line." UCSB Science Line. N.p., n.d. Web. 30 Nov. 2016, from http://scienceline.ucsb.edu/getkey.php?key=144

5.Cao, Yong. "Efficiently Using GPU Memory." CUDA Application Design and Development (2011): n. pag. Web. 28 Nov. 2016, from http://accel.cs.vt.edu/files/lecture10.pdf

6.Cormen, Thomas H., Charles Eric. Leiserson, and Ronald L. Rivest. "String Matching." Introduction to Algorithms. 3rd ed. Cambridge, MA: MIT, 1990. 985-1013. Print.