Efficient GPU Implementation of Bioinformatics Applications

Nuno Miguel Trindade Marcos
Instituto Superior Técnico
November 2014

Abstract— **Biological sequence data is becoming more accessible to researchers around the world. In particular, rich databases of protein and DNA sequence data are already made available to biologist and their size is increasing every day. However, all this obtained information needs to be processed and classified. Several bioinformatics algorithms, such as the Needleman-Wunsch and the Smith-Waterman algorithm, have been proposed for this purpose. Both consist on the execution of dynamic programming schemes which allow the usage of parallelism to achieve a better performance execution. Under this context, this thesis proposes the integration of two previously presented parallel implementations: an adaptation of SWIPE implementation, for multi-core CPUs that exploits SIMD vectorial instructions, and an implementation of the Smith-Waterman algorithm for GPU platforms (CUDASW++ 2.0). Accordingly, the presented work offers an unified solution that tries to take advantage of all computational resources that are made available in heterogeneous platforms, composed by CPUs and GPUs, by integrating a convenient dynamic load balancing layer. The obtained results show that the attained speeup can reach values as high as 6x, when executing in a quad-core CPU and two distinct GPUs.**

Keywords— **Bioinformatics Algorithms; Sequence Alignment; Smith-Waterman Algorithm; Heterogeneous Parallel Architectures; Load Balancing; CUDA.**

I. INTRODUCTION

A. Motivation

Nowadays, numerous databases spread all over the world hosting impressive amounts of biological data, and they are growing in size exponentially as genomes of other species are being sequenced. Specifically, rich databases of protein and DNA sequence data are available on the Internet. One of the most known Bioinformatics algorithms is the Smith-Waterman algorithm [2]. This algorithm is presented in Section III-B and that consists in the alignment of two sequences, using a matrix approach. The problem is that the complexity is $O(n^2)$. So, recently it were presented several parallel computer architectures exploiting some of the parallel approaches: multiple cores processors; multiple processors installed in a single motherboard; multiple computers connected through a common network - cluster on computer grids [6].

B. Objectives

Considering the constant challenges of the growth of this biological data, the aim of the present work is considering two of the Smith-Waterman algorithm implementations, implement a unified solution that tries to take advantage of all computational resources that are made available in heterogeneous platforms, composed by CPUs and GPUs, by integrating a convenient dynamic load balancing layer. With this load balancing layer it is expected that the execution time for the multiple workers considered be equal in the timeline execution. This way it is possible to minimize the waiting times between working and guarantee that all the workers finish their work at the same time. Besides the load balancing layer, it was also proposed several implementations on the GPU module.

II. PARALLEL ARCHITECTURES

According to Almasi et al. [7], a parallel computer architecture is "A collection of processing elements that communicate and cooperate to solve large problems faster". There are several types of parallel computing architectures taking into account different memory organizations, communication between processors/cores and the processor execution model.

According to the several possible approaches, four different types of parallelism may be defined:

- **Bit level**: consists in increasing the size of the computer word, leading to an increase in the amount of information that the processor is able to process in each clock cycle;
- **Instruction level**: enables more than one instruction to be issued and executed by the processor in parallel, as used in multi-stage instruction pipelines and superscalar processors;
- **Data level**: focuses on distributing the data into different computing nodes where the datasets are processed in parallel;
- **Task/Thread level**: characterizes a parallel program where entirely different calculations can be performed on either the same or different sets of data.

A. Flynn's Taxonomy

In 1966 Michael J. Flynn proposed a simple model taking into account the parallelism in instruction execution and memory data calls, based on the idea that both the program and the data memory can be simultaneously addressed by the machine [8]. Flynn's Taxonomy and can be divided into four categories [9]:

1) **SISD**: Single instruction stream, single data stream
2) **SIMD**: Single instruction stream, multiple data streams
3) **MISD**: Multiple instruction streams, single data stream
4) **MIMD**: Multiple instruction streams, multiple data streams

	Single Instruct.	**Multiple Instruct.**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

TABLE I: Flynn's Taxonomy

B. CPU

The Central Processing Unit (CPU) is the computer hardware unit responsible for interpreting and executing the program instructions. One of the first commercial CPU microprocessor was the Intel 4004 presented by Intel in 1971. A CPU is usually composed of these components [10] an Arithmetic Logic Unit (ALU) Responsible for the execution of logical and arithmetic operations; a Control Unit – Decodes instructions, gets operands and controls the execution point; some Registers – Memory cells of the CPU that stores the data needed by the CPU; and CPU interconnection - communication channels among the control unit, ALU, and registers.

C. GPU

A Graphics Processing Unit (GPU) is the processing unit that is present in every graphics card on each computer. This unit is designed specifically for performing the complex mathematical and geometric calculations that are necessary for graphics rendering. The used NVIDIA GPU is a GK110 NVIDIA architecture. This architecture GPU has several Graphics Processing Clusters (GPC)[1], organized on a scalable array. Each GPC contains several Streaming Multiprocessor (SMX)s which perform the executions and run the Compute Unified Device Architecture (CUDA) kernels. The design of the SMX has been evolving rapidly since the introduction of the first CUDA-capable hardware in 2006, with four major revisions, codenamed Tesla, Fermi, Kepler and Maxwell [11]. Kepler's new Streaming Multiprocessor, called SMX, has significantly more CUDA Cores than the SM of Fermi GPUs. Each SMX contains thousands of registers that can be partitioned among the threads under execution, several caches and warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute and execution cores for integer and floating-point operations [12]. A GPU is connected to a host through a high speed IO bus slot (a PCI-Express bus in current systems). The considered GPU model contains four GPCs, fifteen SMXs and six 64-bit memory controllers[2]. In addition to the shared memory, each SMX is composed of [11]: thousands of registers that can be partitioned among threads of execution; several kind of memory caches (explained below); warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute and Execution cores for integer and floating-point operations.

Memory

Specifically, the types of memory presented in a GT100 architecture North American Company that invented the GPUs in 1999 (NVIDIA) GPU are [13]:

- **Global memory**: the largest one (typically greater than 1 GB), but with high latency, low bandwidth, and is not cached. The effective bandwidth of global memory depends heavily on the memory access pattern (e.g. coalesced access generally improves bandwidth).
- **Local memory**: readable and writable per-thread memory with very limited size (16 kB per thread) and is not cached. Access to this memory is as expensive as access to global memory.
- **Constant memory**: read-only memory with limited size (totally 64 kB) and cached. The reading cost scales with the number of different addresses read by all threads. Reading from constant memory can be as fast as reading from a register (e.g. if all threads of a half-warp read the same address).
- **Texture memory**: read-only memory and it is mapped and allocated in global memory. This memory can be used like a cache.
- **Shared memory**: fast on-chip memory of limited size (16 kB per block), readable and writable on a per-block basis. This memory can only be accessed by all threads in a thread block and is divided into equally-sized banks that can be accessed simultaneously by each thread. Accessing this memory is as fast as accessing a register as long as there are no bank conflicts.
- **Registers**: readable and writable per-thread registers. These are the fastest memory to access, but the amount of registers is limited.

D. CPU vs GPU

With the observed increase of the computational demands imposed by the gaming market, the manufacturers of GPUs had to grant powerful processing units, in order to allow gamers to run their increasingly graphically demanding games. A direct consequence is the fact that it represents the most powerful and cost efficient computer hardware. Consequently GPUs are no longer exclusively applied with the purpose of showing computer graphics. An increasingly interest of researchers and developers in the potential of GPUs for applications with large amounts of computations have arisen along the last few years. Today, CPUs in consumer devices have between one and eight cores in a chip, and each of them has some ALUs that perform the arithmetic and logical operations. In comparison, the GPUs have hundreds of cores, each one with four units: one floating point unit, a logic unit, a move or compare unit and a branch unit. An advantage of GPUs is the ability to perform multiple simultaneous operations, up to an order of magnitude of 10^2, since there are hundreds of execution cores in a single GPU. According to Owens et al. [14], one of the major architectural differences between CPUs and GPUs is the fact that CPUs are optimized to achieve high performance in sequential code, with some of the processing stages dedicated to extracting instruction-level

[1]also known as Streaming Processorss (SPs)

[2]`http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf`

parallelism with techniques such as branch prediction and out-of-order execution. On the other hand, GPUs with entirely parallel computing nature allow processing stages to be more focused on computing. This allows achieving a higher level of arithmetic intensity, with around the same number of transistors than CPUs.
Regarding the execution performance, one of the metrics that has been used is floating-point operations per second (FLOPS). During the last years GPUs surpassed CPUs in this measure of theoretical peek performance.

Advantages:

- Faster and Cheaper;
- Fully programmable processing units that support vectorized floating-point operations[15];
- Extremely flexible and powerful, with the introduction of new capabilities in modern GPUs, just like high level languages support the programmability of the vertex and pixel pipelines. Other features are the implementation of vertex texture access, the full branching support in the vertex pipeline, and the limited branching capability in the fragment pipeline.

Disadvantages:

- Memory transfers between host and device can slow all the application;
- Complex memory management, since there are several limitations in regard to memory size (which is limited), and also in the memory organization which has a hierarchical organization.
- Only applications with great parallelization sections can benefit from all the GPU execution power.

E. CUDA - Compute Unified Device Architecture

Definition and Architecture The Compute Unified Device Architecture (CUDA) is a parallel-programming model and software environment designed by NVIDIA, in order to deliver all the performance of NVIDIA's GPUs technology to general purpose GPU Computing. This programming model implements a MIMD parallel processing paradigm, since it divides the execution flow between groups, with the result that every group is independent from another. Inside that group, an adapted SIMD parallelism is adopted, named single instruction, multiple-thread (SIMT), where many threads execute each function. In CUDA, the GPU is denoted as the "device" and the CPU is referred as the "host". "kernel" refers to the function that runs on the device. Using this nomenclature, the host invokes kernel executions in the device.
Current NVIDIA's graphics card are composed of some streaming multi-processors, being all executed in parallel. This execution is done according to a special execution flow (explained in Section II-E). On Kepler, each multiprocessor has 192 processing cores, while on Fermi each multiprocessor has a group of 32 SPs. The high end Kepler has 15 multiprocessors, for a total of 2880 cores ($15 * 192$), and the Fermi accelerators have 16 multiprocessors, for a total of 512 cores ($32 * 16$). Another difference is the shared memory size. On Kepler, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache just like the Fermi GPUs.

Programming Model
CUDA is a extension of C programming language with some reserved keywords. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The __global__ keyword declares a function as being a kernel, and it is executed on the device and can only be invoked by the host using a specific syntax configuration - <<<...>>>. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in $threadIdx$ variable. This kernel functions must be highly parallelized, in order to obtain maximum efficiency for the application [12].

The basics entities involved in the execution of the heterogeneous programming model are the host, which is traditionally the CPU, and the other one is the devices, which are GPUs in this case. The execution flow for a simple CUDA application can be Allocate device memory; Memory copy from host to device; Kernel invocation; Memory copy from device to host and Free device memory.

Execution Model
In CUDA architecture, threads represent the fundamental flow of parallel execution and are executed by core processors [16]. A set of threads is called a thread block. Thread blocks are executed on multi-processors and do not migrate over multi-processors. Several concurrent thread blocks can reside on one multi-processor. This number is delimited by multi-processor resources (shared memory and register file). Finally, a set of thread blocks is called a grid. One kernel is launched as a grid.

III. Sequence Alignment in Bioinformatics

Sequence alignment is a fundamental procedure in Bioinformatics, specifically used for molecular sequence analysis, which attempts to identify the maximally homologous subsequences among sets of long sequences [2]. In the scope of this thesis, it will be considered the processing of biological sequences consisting on a single, continuous molecule of nucleic acid or protein [3]. When comparing sequences, one looks for patterns that diverged from a common ancestor by a process of mutation and selection. The considered mutational processes involved in the alignments are residue substitutions, residue insertions, and residue deletions. Insertions and deletions are common referred to as gaps [17].
The basic idea in the aligning process of two sequences (of possibly different sizes) is to write one on top of the other and break them into smaller pieces by inserting spaces in one

[3]http://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/BIOSEQ.HTML

or the other so that identical sub-sequences are eventually aligned in a one to one correspondence. Naturally, spaces are not inserted in both sequences in the same position.

A. *Optimal Alignment Algorithms*

The optimal alignment of two DNA or protein sequences is the alignment that maximizes the sum of pair-scores less any penalty for the introduced gaps [17].

Optimal Alignment algorithms include Global Alignment algorithms, which align every residue in both sequences. One example is the Needleman-Wunsch algorithm; Local Alignment algorithms, which only considers part of the sequences and obtains the best sub-sequence alignments or the Identification of common molecular subsequences [2].

B. *Smith-Waterman Algorithm*

In 1981, Smith and Waterman [2] proposed a dynamic programming algorithm[4] that computes the similarity scores corresponding to the maximally homologous subsequences among sets of long sequences. Given two sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, the goal of this algorithm is to return a alignment matrix H which indicates the optimal local-alignments between both sequences. For each cell, this algorithm computes the similarity value between the current symbol of sequence A and the current symbol of sequence B. This algorithm has some data dependencies, since each cell of the alignment matrix depends on its left, upper and upper-left neighbors.

Receiving the sequences A and B as input, this algorithm begins with the initialization of the first column and the first row, which is given by:

$$H_{k0} = H_{0l} = 0, \text{ for } 0 \leq k \leq n \text{ and } 0 \leq l \leq m \tag{1}$$

Then the algorithm computes the similarity score $H(i, j)$ by using the following equation:

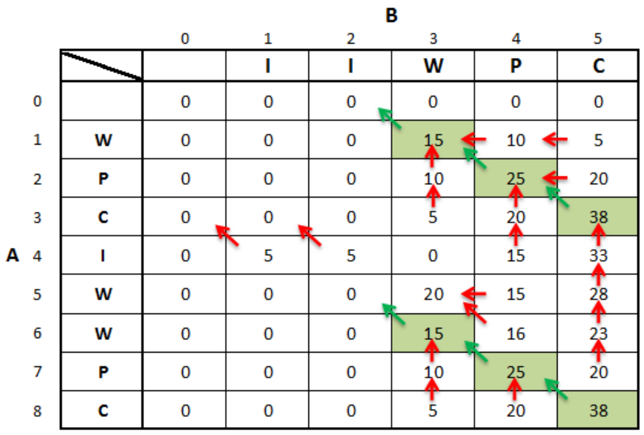
$$H_{ij} = max \begin{cases} H_{i-1_{j-1}} + s(a_i, b_j), & \text{if } a_i \\ \text{ and } b_j \text{ are similars} & \\ H_{i-k_j} - W_k, & \text{if } a_i \\ \text{ is at the end of a deletion of length } k & \\ H_{i_{j-k}} - W_l, & \text{if } b_j \\ \text{ is at the end of a deletion of length } l & \\ 0 & \end{cases} \tag{2}$$

The output for the algorithm is the optimal local alignment of sequence A and sequence B with maximum score.

In order to get all the optimal local alignments between sequences A and B, a trace-back algorithm starts from the highest score in the whole matrix and ends at a score of 0.

Figure 1 presents the optimal local alignments between sequence A: **WPCIWWPC** and sequence B: **IIWPC**. In this example, it is used the BLOSUM 50 matrix scoring model, in order to get $s(a_i, b_j)$ value. The gap penalty is -5. The optimal local alignments between sequences A and B are represented inside green background color cells. These alignments occurred between the subsequences WPC of A and WPC of B.

[4]Dynamic programming is a programming method that solves problems by combining the solutions to their subproblems[18].



		0	1	2	3	4	5
			I	I	W	P	C
0		0	0	0	0	0	0
1	W	0	0	0	15	10	5
2	P	0	0	0	10	25	20
3	C	0	0	0	5	20	38
4	I	0	5	5	0	15	33
5	W	0	0	0	20	15	28
6	W	0	0	0	15	16	23
7	P	0	0	0	10	25	20
8	C	0	0	0	5	20	38

Fig. 1: Smith-Waterman alignment matrix example

C. *CPU Implementations*

In this section we survey the state of the art on CPU-based implementations of the Smith-Waterman algorithm.

Wozniak

One of the first parallel implementations of the Smith-Waterman algorithm was presented in 1997 by A. Wozniak [19], who proposed using SIMD instructions for the parallelization of the algorithm. By exploiting the use of specialized video instructions. These instructions, SIMD-like in their design, make possible parallelization of the algorithm at the instruction level. Another optimization of this implementation is using Visual Instruction Set (VIS) instructions found in the SUN ULTRA SPARC processors. These VIS instructions can be used to execute in parallel four rows of the algorithm, enabling data-level parallelization. VIS instructions use special 64-bit registers, making it possible to add two sets of 16-bit integers and get four 16-bit results, with a single instruction.

Farrar

In order to optimize the performance of the original Smith-Waterman algorithm, Michael Farrar also proposed in 2006 [20] a SIMD solution to parallelize the algorithm at the data level. This solution takes advantage of three different optimizations. The first one is called query profile and was presented by Rognes and Seeberg. It avoids calculating the score between both sequence residues in the Smith-Waterman matrix, calculating a query profile parallel of the query for each possible residue. Then, the calculation of the H_{ij} requires just an addition of the pre-calculated score to the previous H_{ij}. The query profile is stored in memory on 16-byte boundaries. By aligning the profile on a 16-byte boundary, the values are read with a single aligned load instruction, which is faster than reading unaligned data. Another optimization proposed by Farrar is the use of the SSE2 instructions, available on Intel processors. To maximize the number of cells calculated per instruction, the SIMD SSE2 registers are divided into their smallest unit possible. The 128-bit wide registers are divided into 16 8-bit elements for processing. One instruction can therefore operate on 16 cells in parallel. Dividing the register into 8-bit elements limits the cell's range to between 0 and 255. In most cases, the scores fit in the 8-bit range unless the sequences are long and similar. If a query's score exceeds the cells maximum, that query is recalculated using a higher precision. Finally, Farrar proposed the Lazy F evaluation. In order to avoid calculating every cells on the matrix, this optimization makes the algorithm to not calculate the H value when the F remains at zero (thus not contributing to the value of H). In order to avoid bad results, this optimization has a second pass loop to correct all the matrix cells that were not calculated in the first pass. This second pass loop is executed until all elements in F are less than $H - G_{init}$, being G_{init} the gap open penalty. According to the presented results, this algorithm achieves over 3 billion cell updates per second [20].

SWIPE (Rognes)

Taking into account the Farrar's implementation, in 2011 Torbjørn Rognes proposed SWIPE, an efficient parallel solution based on SIMD instructions [3], which allows running the Smith-Waterman search more than six times faster. SWIPE[5] performs rapid local alignment searches in amino acid or nucleotide sequence databases. SWIPE compares sixteen residues from sixteen different database sequences in parallel to the same query residue. This operation is carried out using Intel SSE2 vectors consisting of sixteen independent bytes. Another important characteristic of this algorithm is the use of a compact code of ten instructions written in assembly, which constitute the core of the inner loop of the computations. These ten instructions compute in parallel the values for each vector of 16 cells in independent alignment matrices. The exact selection of instructions and their order is important; this part of the code was therefore hand coded in assembler to maximize performance. In this figure, H represents the main score vector. The H vector is saved in the N vector for the next cell on the diagonal. E and F represent the score vectors for alignments ending in a gap in the query and database sequence, respectively. P is the vector of substitution scores for the database sequences versus the query residue q (see temporary score profiles below). Q represents the vector of gap open plus gap extension penalty. R represents the gap extension penalty vector. S represents the current best score vector. All vectors, except N are initialized prior to this code. Using a 375-residue query sequence, SWIPE achieved 106 billion cell updates per second (GCUPS) on a dual Intel Xeon X5650 six-core processor system, which is more than six times faster than software based on Farrar's approach (the previous fastest implementation).

[5]http://dna.uio.no/swipe/

Pedro Monteiro's Implementation

Extending the Rognes implementation, Pedro Monteiro [4] proposes an inter-task SIMD solution, which extends the previously presented thread-level parallelization model, exploring an fine-grained parallelization approach at inter-task level, exploring both intra and inter-task level parallelization. Basically this implementation consists in the addition of a master/worker model in the original Rogne's SWIPE solution, and creating the concept of processing block. This way, the system has a master thread, represented by the main cycle of the application and has several workers, each one assigned to a physical CPU core. This implementation also attained a performance of more than 71 GCUPS by using 32 parallel worker threads on a distributed-memory architecture, which is nearly 2.5 times faster than SWIPE, running on a different memory architecture [4].

D. GPU Implementations

We now present some of the GPU-based implementations of the Smith-Waterman algorithm found in the literature.

Manavski's Implementation

In order to get a fast implementation of the Smith-Waterman algorithm on commodity GPU hardware using OpenGL instructions, Manavski et al. [21] proposed what they refer to "the first solution based on commodity hardware that efficiently computes the exact Smith-Waterman algorithm". In this implementation, they used an optimization of the Smith-Waterman algorithm previously proposed by Rognes and Seeberg [3]. This optimization consists in pre-computing the query profile parallel to the query sequence for each possible residue, in order to avoid the lookup of $s(a_i, b_j)$ in the internal cycle of the algorithm. Thus, the random accesses to the substitution matrix are replaced by sequential ones. In their implementation, the query profile is stored in GPU texture memory space, since it is a low latency memory. The strategy that was adopted in this implementation consists in making each GPU thread compute the whole alignment of the query sequence with one database sequence. Before that, the database is ordered and stored in the global memory of the GPU, while the query-profile is saved into texture memory. Another optimization of this implementation is the inclusion of an initialization process, where the number of available computational resources is automatically detected. This number will help achieve dynamic load balancing. After this step, the database is divided into as many segments as the number of stream-processors present in the GPU. Each stream-processor then computes the alignment of the query with one database sequence.

To analyze the obtained performance, Manavski's implementation was compared with three previous implementations. This performance was measured by running the application both on single and on double GPU configurations. The first comparison that was carried out is with Liu's implementation of the Smith-Waterman algorithm based in OpenGL instructions. The obtained results show that this implementation is 18 times faster than Liu's one [22]. The second comparison was made with BLAST and SSEARCH algorithms. The obtained results show that this implementation is up to 30 times faster than SSEARCH, and up to 2.4 faster than BLAST. Finally, the last test compares this implementation with Farrar's implementation, showing a three-fold performance increase.

CUDASW++

Just like the algorithm presented above, CUDASW++ is an optimized implementation of the Smith-Waterman algorithm using CUDA. It was proposed by Liu et al. [23] and uses the computational power of CUDA-enabled GPUs to accelerate Smith-Waterman algorithm sequence database searches. Liu et al. presented two different approaches for the parallelization of the algorithm: inter-task parallelization and intra-task parallelization. In inter-task parallelization, each task is assigned to exactly one thread and $dimBlock$ tasks are performed in parallel by different threads in a thread block. In Intra-task parallelization, each task is assigned to one thread block and all $dimBlock$ threads in the thread block cooperate to perform the task in parallel, exploiting the parallel characteristics of cells in the minor diagonals.
In order to achieve the best performance, their implementation uses two stages. The first stage exploits inter-task parallelization and the second stage exploits intra-task parallelization. The transition between these stages is separated by a defined threshold; only when the query sequence length is above that threshold the alignments are carried out in the second stage. Besides this two-stage process, their implementation uses three techniques to improve the performance: coalesced subject sequence arrangement, coalesced global memory access and cell block division method.

Coalesced subject sequence arrangement - For inter-task parallelization, arrange the sorted subject sequence in an array, where the symbols of the query sequences are restricted to be stored in the same column from top to bottom and all sequences are arranged in increasing length order from left to right and top to bottom in the array. For the intra-task parallelization, the sorted subject sequence are sequentially stored in an array, row by row, from the top-left corner to the bottom-right corner. All symbols of a sequence are restricted to be stored in the same row from left to right. Texture cache can be utilized in order to achieve maximum performance on coalesced access patterns.

Coalesced global memory access - This technique explores memory organization patterns in order to achieve the best performance. All threads in a half-warp should access the intermediate results in coalesced pattern. Thus, the words accessed by all threads in a half-warp must lie in the same segment. To achieve this, all threads in a half-warp are allocated in the form of an array to keep them in contiguous memory address.

Cell block division method - This method consists of dividing the alignment matrix into cell blocks of equal size for inter-task parallelization.

When executing their implementation using a single-GPU version, CUDASW++ [23], achieves a performance value of about 10 GCUPS on an NVIDIA GeForce GTX 280 graphics card. In a multi-GPU version, it achieves a performance of up to 16 GCUPS on an NVIDIA GeForce GTX 295 graphics card, which has two G200 GPU-chips on a single card.

Meanwhile, the same authors have proposed a new version of this implementation, the CUDASW++2.0 [5]. In this new version, they proposed three different implementations: a optimized SIMT SW algorithm version, a basic vectorized SW algorithm and a partitioned vectorized SW algorithm.

Optimized SIMT SW algorithm - This implementation is a optimized version of CUDASW++ focused on its first stage, with the introduction of two optimizations: introduction of a sequential query profile and the utilization of a packed data format. The packed data format is used in the re-organization of each subject sequence; four successive residues of each subject sequence are packed together and represented using the uchar4 vector data type. When using the cell block division method, the four residues loaded by one texture fetch are further stored in shared memory for the use of the inner loop.

Basic Vectorized SW algorithm - This implementation is based on Michael Farrar striped SW implementation [20]. It directly maps Farrar's implementation onto CUDA, based on the virtualized SIMD vector programming model. As seen before, Farrar denotes as F values, that part of the similarity values for $H(i, j)$, which derive from the same line: $H_{i-k_j} - W_k$. The lazy-F loop technique avoids the calculations of similarity scores when running this algorithm. This technique states that, for most cells in the matrix $H(i, j)$, the value of $H_{i-k_j} - W_k$ remains at zero and does not contribute to the value of H. Only when H is greater than W_k will F start to influence the value of H.

For the computation of each column of the alignment matrix, the striped SW algorithm consists of two loops: an inner loop calculating local alignment scores postulating that F values do not contribute to the corresponding H values, and a lazy-F loop correcting any errors introduced from the calculations of the inner loop. This algorithm uses a striped query profile.

Partitioned Vectorized SW algorithm - In this implementation, the algorithm first divides a query sequence into a series of non-overlapping, consecutive small partitions, according to a pre specified partition length. Then, it aligns the query sequence to a subject sequence, partition by partition, considering each one a new query sequence. Finally, it constructs a striped query profile for each partition.

Concerning performance evaluation, just like in the first version of CUDASW++ implementation, Liu et al. use two different approaches: a single GPU implementation (NVIDIA Geforce GTX 280) and a multi-GPU implementation (Geforce GTX 295). The optimized SIMT SW algorithm achieves an average performance of 16.5 GCUPS on Geforce GTX 280. The same algorithm, when running on GTX 295, achieves an average performance of 27.2 GCUPS. The partitioned vectorized algorithm achieved an average performance of 15.3 GCUPS using a gap penalty of 10-2 k (gap penalty initialization of 10 and gap extension penalty of 2), gap ; an average performance of 16.3 GCUPS using a gap penalty of 20-2 k; and an average performance of 16.8 GCUPS using a gap penalty of 40-3 k on GTX 280. The same partitioned vectorized algorithm, when running on GTX 295, achieved an average performance of 22.9 GCUPS using a gap penalty of 10-2 k; an average performance of 24.8 GCUPS using a gap penalty of 20-2 k; and an average performance of 26.2 GCUPS using a gap penalty of 40-3 k. When comparing this algorithm with the first CUDASW++ implementation, the optimized SIMT algorithm method runs 1.74 faster on GTX 280 and 1.72 faster on GTX 295. The partitioned vectorized algorithm method runs about 1.58 and 1.77 times faster on GTX 280 and about 1.45 and 1.66 times faster on GTX 295.

IV. Heterogeneous Parallel Alignment MultiSW

Considering two of the best solutions presented: CUDASW++2.0 by Liu et al. [5] and the Pedro Monteiro's SWIPE extension [4], our work proposes an efficient solution for parallel implementation of the Smith-Waterman algorithm, named MultiSW. This implementation consists in the orchestration of both applications execution modules in a single solution, exploiting the use of multiple CPU cores and the NVIDIA GPUs that may be available on the running machine, in a heterogeneous approach methodology like presented in Figure 2.

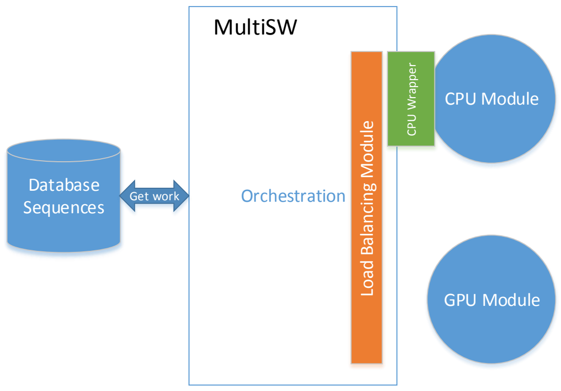


Fig. 2: MultiSW Architecture.

The MultiSW application considers a load balancing abstraction layer, in order to efficiently split the database sequences during the execution. Another implemented optimization is a wrapper[6] function for the CPU worker execution. These additional implementations were proposed in order to improve the application CPU worker execution time.

[6]A wrapper function is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine or a system call with little or no additional computation.

A. *Implementation Details and Optimizations*

Database File Format As for the database file format, Pedro Monteiro's implementation [4] only considers the BLAST sequences type, and the CUDASW++ only considers the FASTA format. So, all the functions used from Pedro Monteiro's implementation were adapted to use the FASTA database sequence file format.

Dynamic Load-balancing Layer

"Load balancing is dividing the amount of work that a computer has to do between two or more computers so that more work gets done in the same amount of time and, in general, all users get served faster" [7]. So, in our work, the processing unit represents the database sequences, that needs to be aligned against the query sequence. The execution time of each worker iteration is directly affected by the size of the processed block. So to make the implementation more efficient and to adjust the execution time for all workers, in this implementation, its also considered a load balancing module, adjusting dynamically the block size of the obtained work. The load balancing layer dynamically adjusts the block size for each worker (concept of block size is presented above). In this implementation, all workers were considered equals. The only difference is that the default CPU worker block size was 30000 and the GPUs default block size is 65000. Imagine the scenario presented in Figure 3. In this case, the worker A spends almost twice as much time than the time that worker B takes to process its block. If the application finishes its execution after the worker B finishes its iteration, the worker A has not been processing all the information. This way, the solution does not take advantage of each worker the efficient possible way.

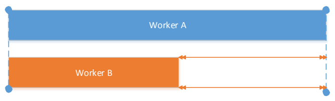


Fig. 3: Workers execution not balanced.

In order to minimize this inefficiency in each iteration, the proposed load balancing layer adjusts the block size of each worker, to make the execution time as close as possible. Considering this, it is meant to adjust the block size, in order to reduce the execution time of the worker A.

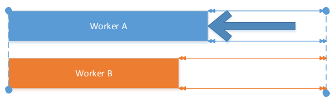


Fig. 4: Workers execution balanced.

In the developed model, the next variables were considered:

- $blocksize(w, i)$ - Represents the block size computed by worker w in iteration i;
- $T_{execution}(w, i)$ - Represents the execution time of worker w in iteration i;

[7]`http://searchnetworking.techtarget.com/definition/load-balancing`

- $T_{minexecution}(i)$ - Represents the minimum execution time for all workers in iteration i.

When a worker finishes its execution, it calls the $registerExecutionTime$ function, that registers for the $deviceNum$ worker the execution time and processed block size. This function updates the attributes for the current worker execution and calls a method named $adjustBlockSizes$ that reprocesses all worker's block sizes.

The first worker (fastest one) to finish their work increases their block size 10% considering the previous block size. So, next iteration block size is going to be:

$$blocksize = blocksize \times 1,1 \tag{3}$$

After all the workers finish their execution, the new block size for each worker is calculated taking into account the fastest worker execution, and the time spent in that execution. This time is presented in Equation 4 and it's given by the function $getMinExecTime()$.

$$T_{minexecution} = getMinExecTime(); \tag{4}$$

Then, for each worker, the block size of next iteration can be calculated by:

$$blockSize(i) = ceil(\frac{T_{minexecution} \times blockSize(i-1)}{T_{execution}(i-1)}) \tag{5}$$

This block size must be an integer values, so we use the ceil function to round the value obtained by the formula.

V. EXPERIMENTAL RESULTS

The performance of our implementation - MultiSW - was evaluated by considering multiple execution scenarios. Mixing both CPU cores and GPUs, the results are presented below in section V-C. The experimental setup was a Linux based workstation with an Intel(R) Core(TM) i7 4770K @ 3.5GHz CPU, Four Kingston HyperX DDR3 CL9 8GB @ 1.6GHz Memory RAM modules, ASUS Z87-Pro Motherboard, an MSI GeForce GTX 780 Ti Gaming 3GB DDR5 and a GeForce GTX 660 Ti 2GB DDR5.

A. Experimental Dataset

The query sequence that was used in the experimental scenarios was the IFNA6 interferon, alpha 6 [Homo sapiens (human)] [24] with 189 residues. The database sequences that were considered was the release 2014_02 of UniProtKB/Swiss-Prot [25] database sequences in the FASTA format repeated 5 times in the file. This database contains 542.503 sequences with several sizes, comprising 192.888.369 amino acids abstracted from 226.190 references. The total processed number of sequences are 2.712.515.

B. Evaluating Metrics

In order to compare the multiple considered scenarios, the speedup metric will be used:

$$speedup = \frac{t_{sequential}}{t_{parallel}} \tag{6}$$

C. Results

This section presents multiple scenarios and their results when running the application in distinct execution parameters configurations. It starts with the simplest scenario, corresponding to a single CPU core execution, and finishes with the most complex configuration with an orchestration of the workers based on a multicore CPU and multiple GPUs, that processes all the available work.

Scenario A - Single CPU core Considering a single CPU core execution, the total execution time was about 31,52 seconds, as shown in Figure 5. The multiple grey colored blocks represents the execution for each CPU wrapper iteration, considering its size of 30.000 sequences. These iteration execution times vary between 0,0688 and 2,391 seconds.

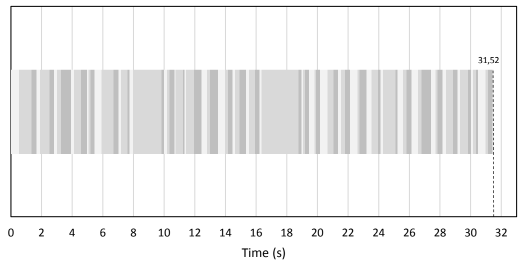


Fig. 5: Processing times considering a single CPU core execution and a processing block with 30000 sequences.

Scenario B - Four CPU cores Considering a four CPU cores execution, the total execution time was about 15,55 seconds, as shown in Figure 6. The distinct grey-based colored blocks represents the process time for a block of 30.000 sequences (considering a CPU wrapper iteration) by four CPU cores. These iteration values vary between 0,046 and 0,957 seconds.

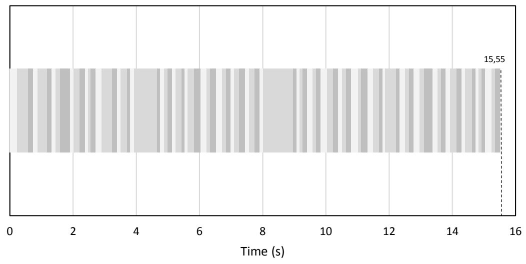


Fig. 6: Processing times for 4 CPU cores, considering a block size of 30.000 sequences.

Scenario C - Single GPU - GeForce GTX 780 Ti Considering a single GeForce GTX 780 Ti GPU execution, the total execution time was about 6,35 seconds, as show in Figure 7. In the figure it is presented several grey colored execution blocks that represents the time of processing 65.000 database sequences against the query sequence. These iteration values vary between 0,118 and 0,266 seconds.

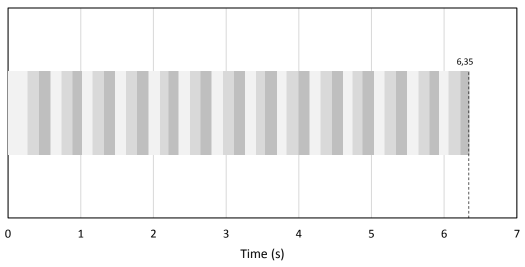


Fig. 7: Processing Times for single GPU in Machine A, considering blocks size of 65.000 sequences. Total execution time about 6,35 seconds.

Scenario D - Single GPU - GeForce GTX 660 Ti Considering a single GeForce GTX 660 Ti GPU execution, the total execution time was about 7,38 seconds, as show in Figure 8. In the figure it is presented several grey colored execution blocks. Each one of them represents the time of processing 65.000 database sequences against the query sequence. These iteration values vary between 0,126 and 0,304 seconds.

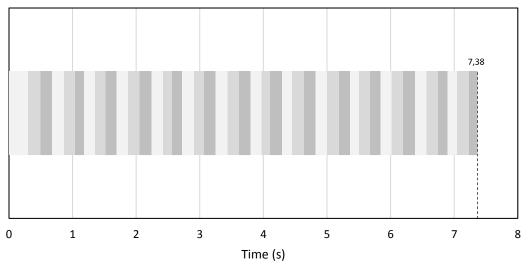


Fig. 8: Processing Times for single GPU in Machine B, considering blocks size of 65.000 sequences. Total execution time about 7,38 seconds.

Scenario E - Four CPU cores + Single GPU Execution In this scenario the considered workers for the execution are the four CPU cores and the GeForce GTX780 Ti GPU. This execution time was about 6,112 seconds, as show in Figure 9. The iteration execution times for the GPU worker varies between 0,082 and 0,316 seconds. For the CPU worker this value goes between the 0,072 and the 0,258 seconds. The execution CPU worker processed 817.087 sequences, while the GPU worker processed 1.895.428 sequences.

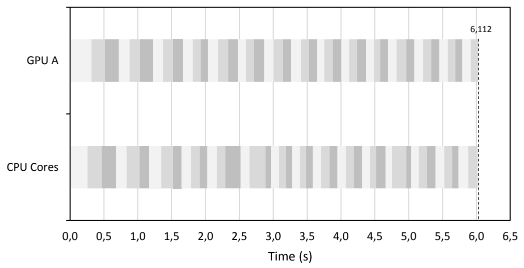


Fig. 9: Processing Times for 4 CPU cores and a GeForce GTX780 Ti GPU, considering CPU blocks of 30.000 sequences and GPU blocks of 65.000 sequences. Total execution time was 6,112 seconds.

In the Figure 9 it is also presented the dynamic block size along the time. These values are presented in the arrow next to the block execution, in the GPU worker and in the CPU worker. So the CPU worker starts with the 30.000 block size and finish with a size of 15.000. The GPU worker starts with a 65.000 sequences block size and finish with a size of 40.000 sequences. To both of the workers, the number of next processing sequences is decreasing along the execution time, since it was the way load balacing module works.

Scenario F - Four CPU cores + Double GPUs Execution The last scenario considered are composed by a four CPU cores execution and both available GPUs: the GeForce GTX 780 Ti (GPU A) and the GeForce GTX 660 Ti (GPU B). Like expected this execution was the fastest one and takes about 4,957 seconds, as show in Figure 10. The CPU worker process 70.795 sequences, the GPU A worker computed 1.241.496 sequences and the GPU B worker process 1.059.202 sequences.

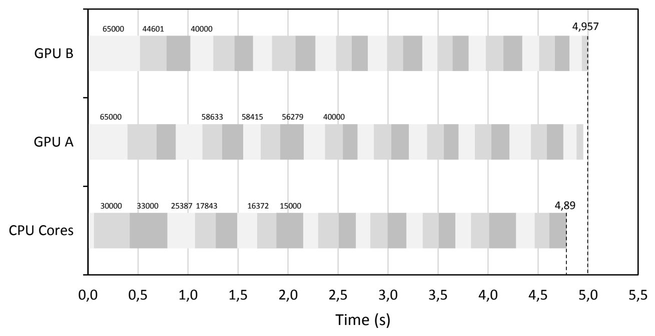


Fig. 10: Processing Times for 4 cores CPU, GPU A and GPU B, considering the initial block size of 30.000 sequences blocks to the CPU solution and 65.000 to the GPU solution. Total execution time of 4,957 seconds. Near some of the iteration blocks it is presented the new considered block size.

VI. Summary

Like it is show in Table II, considering the multiple scenarios, the Scenario F presented in Section V-C was achieved a speedup of 6,4x when comparing the execution in a CPU single core execution presented in Scenario A, Section V-C.

	Execution Time	Speedup
Single core	31,52	
Four cores	15,55	2,03
GeForce GTX 780 Ti	6,350	4,96
GeForce GTX 660 Ti	7,380	4,271
Four CPU cores + GeForce GTX 780 Ti	6,112	5,16
Four CPU cores + 2 GPU	4,96	6,36

TABLE II: Execution Speedups.

The many more workers are considered in the execution of our work orchestration, the greater synchronization between the involved threads are needed. This causes the occurrence of execution delays and makes the workers wait longer. This situation is minimized with the load balancing layer presented in our solution, since the block size is being adapted to be similar. However, there are some limitations in the loading balance module, since the number of total processing sequences are not known in the beginning of the application.

VII. Conclusions and Future Work

Multiple solutions have been proposed in the last years to be possible to respond to the large amount of biological information produced everyday. Exploiting parallel architectures based on CPU and GPU architectures, there has been possible to quickly process these data. So, in our work, it was proposed a solution that mixes both to get better results. Under this context, this thesis proposed the integration of two previously presented parallel implementations: an adaptation of SWIPE implementation [3], for multi-core CPUs that exploits SIMD vectorial instructions [4], and an implementation of the Smith-Waterman algorithm for GPU platforms (CUDASW++ 2.0) [5]. Accordingly, the presented work offers a unified solution that tries to take advantage of all computational resources that are made available in heterogeneous platforms, composed by CPUs and GPUs, by integrating a convenient dynamic load balancing layer. The obtained results presented in Section V show that the attained speedup can reach values as high as 6x, when executing in a quad-core CPU and two distinct GPUs.

A. Future Work

As future work can be add an extra thread to the solution to prepare all the GPU work, like it occurs in the CPU module. With the new Kepler GPUs it is possible to explore the Dynamic Parallelism, which corresponds to the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU. This would be good for bigger sequences to avoid big iteration execution blocks. Besides this possible optimizations, another load balancing algorithm can be also explored and analysed with the solution.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," J. Mol. Biol., vol. 48, pp. 443–453, 1970.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." Journal of molecular biology, vol. 147, no. 1, pp. 195–197, Mar. 1981. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/7265238

[3] T. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation," BMC Bioinformatics, vol. 12, no. 1, pp. 221+, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-12-221

[4] P. M. Monteiro, "Profiling biological applications for parallel implementation in multicore computers," Master's thesis, Instituto Superior Tecnico, Av. Rovisco Pais, 1, November 2012.

[5] Y. Liu, B. Schmidt, and D. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," BMC Research Notes, vol. 3, no. 1, pp. 93+, 2010. [Online]. Available: http://dx.doi.org/10.1186/1756-0500-3-93

[6] D. Culler, J. Singh, and A. Gupta, Parallel computer architecture: a hardware/software approach, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 1999. [Online]. Available: http://books.google.pt/books?id=gftcVOn7iGsC

[7] G. Almasi and A. Gottlieb, Highly parallel computing, ser. The Benjamin/Cummings series in computer science and engineering. Benjamin/Cummings Pub. Co., 1994. [Online]. Available: http://books.google.pt/books?id=rohQAAAAMAAJ

[8] J. Hennessy, D. Patterson, and A. Arpaci-Dusseau, Computer architecture: a quantitative approach, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2007, no. vol. 1. [Online]. Available: http://books.google.pt/books?id=57UIPoLt3tkC

[9] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.

[10] W. Stallings, Computer Organization and Architecture: Designing for Performance. Prentice Hall, 2010. [Online]. Available: http://books.google.es/books?id=-7nM1DkWb1YC

[11] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming. Pearson Education, 2013. [Online]. Available: http://books.google.pt/books?id=ynydqKP225EC

[12] NVIDIA CUDA - NVIDIA CUDA C Programming Guide, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, February 2014. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[13] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st ed. Addison-Wesley Professional, 2010.

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," Computer Graphics Forum, vol. 26, no. 1, pp. 80–113, 2007.

[15] J. Nickolls and W. J. Dally, "The gpu computing era," IEEE Micro, vol. 30, no. 2, pp. 56–69, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.41

[16] B. Oster and G. Ruetsch, "Getting started with CUDA," in NVISION 2008, The World of Visual Computing. NVIDIA, 2008. [Online]. Available: http://www.nvidia.com/content/nvision2008/tech_presentations/CUDA_Developer_Track/NVISION08-Getting_Started_with_CUDA.pdf

[17] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Jul. 1998.

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, Introduction to Algorithms, 2nd ed. McGraw-Hill Higher Education, 2001.

[19] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison." Computer Applications in the Biosciences, vol. 13, no. 2, pp. 145–150, 1997. [Online]. Available: http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics13.html#Wozniak97

[20] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," Bioinformatics, vol. 23, pp. 156–161, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btl582

[21] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," BMC Bioinformatics, vol. 9, no. Suppl 2, p. S10, 2008. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2323659&tool=pmcentrez&rendertype=abstract

[22] Bio-sequence database scanning on a GPU, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1109/ipdps.2006.1639531

[23] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units." BMC research notes, vol. 2, no. 1, pp. 73+, 2009. [Online]. Available: http://dx.doi.org/10.1186/1756-0500-2-73

[24] NCBI, "Ifna6 interferon, alpha 6 [homo sapiens (human)] - ncbi," 2014, accessed on October 9, 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/gene?Db=gene&Cmd=ShowDetailView&TermToSearch=3443

[25] G. D. Database, "Genbank dna database," http://www.ncbi.nlm.nih.gov/genbank/, 2014, accessed on October 9, 2014.