Speeding up Subset Seed Algorithm for Intensive Protein Sequence Comparison

Van Hoa NGUYEN
IRISA/INRIA Rennes
Rennes, France
Email: vhnguyen@irisa.fr

Dominique LAVENIER
CNRS/IRISA
Rennes, France
Email: lavenier@irisa.fr

***Abstract*—Sequence similarity search is a common and repeated task in molecular biology. The rapid growth of genomic databases leads to the need of speeding up the treatment of this task. In this paper, we present a subset seed algorithm for intensive protein sequence comparison. We have accelerated this algorithm by using indexing technique and fine grained parallelism of GPU and SIMD instructions. We have implemented two programs: iBLASTP, iTBLASTN. The GPU (SIMD) implementation of the two programs achieves a speed up ranging from 5.5 to 10 (4 to 5.6) compared to the BLASTP and TBLASTN of the BLAST program family, with comparable sensitivity.**

I. Introduction

The sequence similarity search finds similar segments, or local similarities, between two DNA sequences or protein sequences, measured by match, mismatch and gap scores. To establish similarity, an *alignment score* is calculated. Its objective is to locate similar regions in the DNA or protein sequences because high degree of similarity often implies similar function or structure. A typical search is to query a bank with a gene whose function is unknown. The results of the request correspond to similar segments.

Recent biotechnology improvements in the sequencing area lead to an increasing amount of genomic databases. Genbank [1], for example, contains more than 180 billion of nucleotides (November 2007) and its size is multiplied by a factor ranging from 1.4 to 1.5 every year. Thus, the similarity search is a very time consuming task which may need the use of supercomputers.

Similarity search is primarily a data content search. Banks are systematically scanned from the first to the last sequence. It is a challenging task requiring a huge computing power. With the stagnation of microprocessor frequencies, there is a great interest in optimizing this task.

There are several algorithms to search alignments. One of the first is the Smith-Waterman algorithm developed in 1981 [15]. It uses dynamic programming techniques and has a quadratic complexity. Latter ones, developed in 1990 [12] [13], as the BLAST program family, are based on heuristic. The key of the BLAST heuristic is that a statistically significant alignment is likely to contain a common word of W characters. Thus, instead of scanning the whole search space, the search only focus to these specific common words, called anchor points. Consequently, the computation time is drastically reduced. However, the heuristic does not guarantee to detect the best alignments, but it reports most of the significant alignments.

More recently, several works have proposed to use spaced seeds to perform detection of anchor point, including PatternHunter [10], PartternHunter II [9]. Instead of considering a word of W contiguous characters as a seed, a word of no necessarily consecutive W characters may be considered. More precisely, a spaced seed looks like a binary mask string, such as 10101, where matches in all positions 1 are required for an anchor point; 0 denotes allowed mismatches. The next seed generation doesn't consider one seed but a set of several seeds which can be of different length and possibly made of different spaced seeds (multiple seeds) [8]. Another seed family (subset seed) groups different characters in the same set [6], still providing better expressiveness.

This paper presents an effective implementation of a recently proposed seed-based heuristic, called *subset seed*, that exploits fine-grained parallelism using data indexing for intensive protein sequence comparison, such as the comparison of two protein banks or the comparison of a protein bank and a genome translated to its six reading frames. We use Intel SSE2 SIMD instructions and the SIMD architecture of GPU (Graphics Processing Unit) device to speed up critical section of the algorithm.

The remainder of this paper is organized as follows. In section 2, a brief background of the subset seed algorithm is given. Section 3 describes how to use Intel SSE2 SIMD instructions and SIMD architecture of GPU for implementing this algorithm. In section 4, we present numerical test results before section 5 concluding and presenting future work.

II. Similarity search

In this section we present the subset seed similarity search algorihtm, including seed-based heuristic.

A. Seed based similarity search

The idea of heuristic search is based on an observation: In general, significant alignments have a region of high similarity which is characterized by a zone of W identical characters - word - between two sequences that make up this alignment. Therefore, first, we can locate places where these words appear. When the same word is located on two sequences, it provides an anchor point for calculating alignment by

extending to the right and to the left the number of *match* between identical symbols. If the extension finds a significant similarity (measured in the number of *match* / *mismatch*), a final phase is processed to build a better alignment by considering the inclusion or deletion of symbols (*gap*).

The scheme below shows a construction example of alignment between two protein sequences. The word ARV, in both sequences, serves as an anchor point (stage 1). Starting from this anchor point, we extend to the right and to the left for getting an interesting *match / mismatch* ratio (stage 2). Finally, we extend the alignment for taking into account the errors of *gap* (stage 3).

```
stage #1
   K V I T A R V T G S A Q W C D N T G V K N I H M
           | | |
   K L I S A R V K G S Q F C T N P T G M K A N I H
stage #2
   K V I T A R V T G S A Q W C D N T G V K N I H M
   |   |   | | |   | |           | |   |
   K L I S A R V K G S Q F C T N P T G M K A N I H
stage #3
   K V I T A R V T G S A Q W C D N - T G V K - N I H
   |   |   | | |   | |   |   |   |   | |   |   | | |
   K L I S A R V K G S - Q F C T N P T G M K A N I H
```

This is a 3-stage process: (1) anchor point detection; (2) *ungapped* alignment; (3) *gapped* alignment. This heuristic is implemented with many variants in a number of programs such as programs of the BLAST family which are the reference programs in bioinformatics. Depending on the type of database inputs, each BLAST [12] [13] program has a different name: BLASTP when the query is an amino acid sequence and the database is a protein bank, TBLASTN when the query is an amino acid sequence and the database is a nucleotide bank.

In intensive protein sequence comparison between two banks (bank1 and bank2), BLASTP executes comparison of sequence S (bank1) against the set of sequences in bank2. It first searches for all hits between sequence S and bank2 by using an index structure such as a lookup table. Then, these hits are extended, first without gaps, then allowing them. Hits meeting specified threshold are returned as statistically significant alignments.

The advantage of BLASTP is its speed compared to dynamic programming techniques. The speed depends on the hit size. Actually, longer the hit size, lower the possibility to locate anchor point.

B. Subset seed similarity search

In the context of intensive protein sequence computation of two banks: bank1 (N sequences) and bank2, BLASTP must scan and process bank2 N times. To avoid this drawback, we dynamically index the two banks in main memory. Based on this idea, the two banks are scanned only once just for making the index.

The advantage of a double indexing is that the systematical scan of bank2 disappears. Through an appropriate indexing structure, we can directly point to all identical words in the two banks. If a W-AA word appears $|nb1|$ times in bank1 and $|nb2|$ times in bank2, there are $|nb1| \times |nb2|$ hits. It means that there are also $|nb1| \times |nb2|$ ungapped alignments to calculate.

TABLE I
iBLASTP GENERIC ALGORITHM

```
 1:  index1 = make index (bank1)                    #stage 0
 2:  index2 = make index (bank2)
 3:  for all possible seeds
 4:     construct neighboring - block nb1           #stage 1
 5:     construct neighboring - block nb2
 6:     for each subsequence of nb1
 7:       for each subsequence of nb2
 8:         compute ungapped alignment              #stage 2
 9:         if score ≥ S1
10:           compute small gapped alignment        #stage 3.1
11:           if score ≥ S2
12:             compute full gapped alignment       #stage 3.2
13:             if score ≥ S3
14:               traceback & display alignment     #stage 4
```

The subset seed algorithm (iBLASTP) proceeds in 5 successive stages as shown on TABLE I. Stage 0 index the two banks, stage 1 constructs two neighboring blocks, stage 2 performs ungapped alignments, stage 3 computes gapped alignments and stage 4 displays alignments. We describe in detail these stages in the following subsections.

Stage 0: Bank indexing. Spaced seeds [9] and their relative, vector seeds [8], can augment selectivity and reduce the false positive rate in the seeding step. But spaced seeds can not distinguish between different types of mismatches in the case of protein sequence. Because all amino-acids are not equivalent, we can create groups of characters. Based on the groups, [6] [7] created subset seeds, the extension of spaced seeds. Subset seeds have an intermediate expressiveness between spaced and vector seeds but they allow an efficient hashing method to locate hit. The main advantage of subset seeds is that they provide a powerful seed definition and, in the same time, preserves the possibility of direct indexing. A subset seed of W characters (see detail in [3]) is defined as a word $(s_1 s_2 ... s_W)$. Figure 1 presents a subset seed example of 4 characters.

```
#  | A, C, D, E , F, G, H, K, I, L, M, N, P, Q, R, S, T, V, W, Y
@1 | c={CFYWMLIV}, g={GPATSNHQEDRK}
@2 | A, C, G, P, f={FYW}, i={IV}, l={LM}, n={NH}, q={QED}, k={KR}, t={TS}
#  | A, C, D, E , F, G, H, K, I, L, M, N, P, Q, R, S, T, V, W, Y
```

Fig. 1. Subset seed example with length equal to 4

To build indexing structure for bank1, we store all sequences into memory. Then each sequence is parsed into fixed length overlapping words of W characters - W-AA words. Each W-AA word is matched in subset seed for finding a corresponding word. For example, to apply the subset seed of Figure 1, the words AQAA, APAA, ASAA are translated to a similar word $AgAA$ (or a seed). Furthermore, each occurrence of W-AA word has a position (index). We use a buffer integer (size equal to size of bank1) to store all positions of occurrences of similar W-AA words (translated), for example the words AQAA, APAA, ASAA. The positions of similar W-AA words (ex. $AgAA$) are considered as a list of W-AA words. We have 20^W (20 is the number of amino acids) lists corresponding to

20^W W-AA words. The number of linked lists depends on the subset seed being used. For example, if $W = 4$, there are 20^4 words: AAAA, ..., YYYY and when applying a subset seed such as in Figure 1, there are 8800 ($20 \times 2 \times 11 \times 20$) linked lists. We use an other buffer to store the heads of the linked lists.

Bank2 is processed in the same way. At the end of this stage, for each W-AA word (in subset seed), we have two lists which content $|nb1|$ occurrence positions of W-AA word in bank1 and $|nb2|$ occurrence positions of W-AA word in bank2.

Stage 1: Construct two neighboring blocks. Hits detected in the first stage are considered as the base of ungapped alignments. The objective is to be able to rapidly decide if one hit has favorable environment to build an alignment. The indexing structure above gives us the lists of positions of W-AA words and their neighboring knowledge for performing ungapped alignments. In this stage, for each W-AA word, we construct two neighboring blocks: one block (nb1) for bank1 and another block (nb2) for bank2. To build neighboring block nb1, index1 provides a list of all W-AA word positions, then for each position a subsequence is copied. Each subsequence contents W-AA word and its neighboring knowledge. Two blocks of subsequences allow us to reuse cache memory when we perform ungapped alignment.

Stage 2: Ungapped alignments. For each W-AA word, we perform $|nb1| \times |nb2|$ ungapped alignments of two neighboring blocks. More precisely, for each subsequence in block nb1, we calculate the ungapped score with $|nb2|$ subsequences in block nb2. Ungapped extension is performed on subsequences of bounded length, and ignores insertion and deletion events. The aim is to quickly compute a score according to a substitution-score matrix. Starting from 0, the score is increased or decreased depending on the substitution-score values in this matrix. If ungapped extension score is superior than a specified threshold S_1, this ungapped alignment is passed to the next stage.

Stage 3: Gapped alignments. We use the dynamic programming technique to build gapped alignments. This stage is divided into two sub-stages: The first computes small gapped alignments (stage 3.1), it aims to limit the searching space of dynamic programming. In this procedure, there is a constraint on the number of allowed gap. If score exceeds a specified threshold S_2, the standard dynamic programming procedure is launched (stage 3-2).

A gapped alignment contents at least one ungapped alignment. If a gapped alignment contents two ungapped alignments, they are not always on the same diagonal. For this reason, sometime, we must recalculate gapped alignments for different ungapped alignments. We can avoid this problem by storing all ungapped alignments contented in the gapped alignments in a list, and performing gapped alignment only for hit which doesn't belong to this list.

stage 4: Traceback & display. In this stage, trace-back information optimizes alignments recorded in previous stage and displays them to user.

The structure of our approach has been developed with objective of exploiting fine grained parallelism on specific hardware structure. Thus, the size of the subsequences in stage 2 and 3-1 is fixed.

C. Profiling of the code

We have implemented two sequential programs: iBLASTP and iTBLASTN by reference to the programs BLASTP and TBLASTN programs of the BLAST family. The first compares two protein banks and the second compares a protein bank with a genome translated into 6 reading frames (6 ways to match a DNA to proteins). Both versions run on standard processors. They allow us to measure more precisely the speed-up of fine grained parallelism.

We have examined the average runtime for each stage. Results are shown in TABLE II. We found that the second and third (3-1) stages consumed respectively 60-72% and 23-25% of the total execution time.

TABLE II
AVERAGE RUNTIME FOR EACH STAGE

stages	0	1	2	3-1	3-2	4
iBLASTP	0,4 %	0,6 %	60 %	25 %	11 %	3%
iTBLASTN	0,4 %	0,6 %	72 %	23 %	2,7 %	1,3%

III. SPEEDING UP SUBSET SEED ALGORITHM USING SIMD ARCHITECTURE

The processing time of ungapped and small gapped alignment takes most of the time. Obviously, to get a significant speed up, execution time of these stages needs to be reduced. To increase the performance of these stages, we exploit two types of fine-grained parallelism: SIMD instructions of modern microprocessors and SIMD architecture of recent GPU.

A. SIMD instructions

With introduction of MMX microprocessors in 1997, Intel made computing with SIMD technology available in a general-purpose microprocessor. Multimedia extensions exploit fine-grained parallelism, where computations are split into sub-words with independent units operating on them simultaneously. SIMD processing was enhanced with the addition of the SSE2 (Streaming SIMD Extension) in the Pentium 4 microprocessor. It permits handing sixteen simultaneous byte operations in 128-bit XMM registers.

1) Ungapped alignments: Instead of performing $|nb1| \times |nb2|$ ungapped alignments for a W-AA word by using the Single Instruction Single Data (SISD), we have used SIMD instructions. In this case, a score fits into 1 or 2 bytes instead of 4 bytes.

The SIMD register simultaneously contains k scores related to k subsequences of block nb1. When k subsequences are computed with all subsequences of block nb2, a simple speed improvement is achieved by creating a kind of profile score for the k subsequences. This profile score is a specific substitution-score matrix of k subsequences. Instead of indexing the original substitution-score matrix by subsequences symbols, the

new matrix is indexed by the position on the k subsequences and a subsequence symbol (block nb2).

This profile score permits to load substitution-scores in a single read operation. When using 1 byte integer k = 16 and when using 2 byte integer k = 8. Each element of the SIMD registers maps one subsequence. The first element maps $s1\_1$, the second element maps $s1\_2$, still the last element maps $s1\_k$. Thus, score(i) (between k sequences in block nb1 and one subsequence of block nb2) is considered as a score result between the i-th subsequence and the subsequence in block nb2.

When k = 16, the 128-bit wide registers are divided into 16 8-bit elements. Dividing the registers into 8-bit elements limits the range of scores from 0 to 255. In mots cases, the scores fit in the 8-bit range unless the subsequences are totaly similar.

```
          block nb1                          block nb2
s1_1: ..K C [A] G A A S D..   s2_1: ..N M [A] H A A S Q..
s1_2: ..A G [A] S A A V N..
s1_3: ..L N [A] A A A W E..
s1_4: ..Y E [A] N A A L T..
s1_5: ..M T [A] S A A K H..
s1_6: ..S H [A] G A A S Q..
s1_7: ..W Q [A] D A A T S..
s1_8: ..T S [A] E A A E M..
```

The above scheme illustrates an ungapped alignment computation between 8 subsequences in block nb1 with one subsequence in block nb2 when the position on 8 subsequences is at the beginning of seed and subsequence symbol (s1 of black nb2) is "A".

2) Small gapped alignments: In addition, we also use SIMD instructions to compute small gapped alignments. The ungapped extensions passed to the stage 3.1 are stored in a list. When this list contains at least K elements, all elements on this list are considered for small gapped alignment with SIMD instructions.

Unlike ungapped alignment, if there is one ungapped alignment passed equally there is a pair of subsequences. The pair of subsequences in small gapped alignment is usually similar. Furthermore, the length of subsequence is longer than the length of subsequence in ungapped alignment. Consequently, when k =16, scores usually fit out the range of 8-bit. Thus, we decided for a unique value of k = 8.

In addition, we must perform only one small gapped alignment for a pair of subsequences. The computing begins at seed position, extends to the right and then to the left. Thus, we cannot use the subsequence profile because it is just used once. Consequently, in the Smith-Waterman algorithm [15], to compute each cell score, the value of substitution-score must be sequentially calculated. More precisely, symbols from the two subsequences have to be read and look-up into the substitution-score matrix in order to calculate the corresponding substitution-score. This process has to be repeated for the 8 elements in the SIMD register.

The scheme below illustrates small gapped alignment computation for 8 pairs of subsequences. The computing begins at seed position, extends to the right and then to the left. The score of pair(i) is a sum of score(i) of two extentions.

```
          bank1                          bank2
s1_1: V..K S [A] G A A S..V   s2_1: N..A D [A] H A A G..N
s1_2: N..R N [A] S A A F..S   s2_2: G..Q Q [A] G A A F..L
s1_3: K..V R [A] A A A K..M   s2_3: T..G Q [A] N A A K..G
s1_4: F..I V [A] N A A Q..K   s2_4: F..Y G [A] P A A I..T
s1_5: A..R I [A] S G A E..I   s2_5: D..L I [A] K A A E..F
s1_6: G..H R [A] G A A L..F   s2_6: C..H L [A] R A A V..A
s1_7: H..D L [A] D A A E..E   s2_7: Y..N F [A] N A A D..G
s1_8: S..R S [A] E A A Q..A   s2_8: I..G F [A] T A A Q..S
```

B. SIMD architecture of GPU

During the last decade, GPUs [4] have been developed as highly specialized processors for the acceleration of raster graphics. The GPU has several advantages over CPU architectures for highly parallel intensive workloads, including higher memory bandwidth, significantly higher floating-point capabilities, and thousands of hardware thread contexts with hundreds of parallel compute pipelines executing programs in a single instruction multiple data (SIMD) mode.

NVIDIA has introduced a new GPU, i.e. Geforce 8800 GTX and a C-language programming called CUDA [4] (Compute Unified Device Architecture). Geforce 8800 GTX architecture comprises 16 multiprocessors. Each multiprocessor has 8 SPs (streaming processors) for a total of 128 SPs. Each group of 8 SPs shares one L1 data cache. A SP contains a scalar ALU (Arithmetic Logic Unit) and can perform floating point operations. Instructions are executed in a SIMD mode.

The threads on GPU are organized in blocks. The grid of thread blocks is executed on the device. Thread blocks have the same dimensions. A thread block is processed by only one multiprocessor, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses. A multiprocessor can process several thread blocks concurrently by partitioning among the set of registers and the shared memory.

For each W-AA word, $|nb1| \times |nb2|$ ungapped alignments are performed in parallel by the GPU. The ungapped extensions passed to the stage 3.1 are stored in a list. When this list contains at least K elements, all elements on this list are considered for small gapped alignment on the GPU.

1) Ungapped alignments: We implemented ungapped alignment stage on graphic card by using the matrix multiplication algorithm [4]. For each W-AA word, there are two subsequence blocks. Suppose that block A[wA, hA] corresponds to block nb1; wA is the length of subsequences, hA is the number of subsequences in block nb1; block B[wB, hB] corresponds to block nb2, wB is the number of subsequences in block nb2, hB is the length of subsequences. On the other hand, block B is the transposition of block nb2. Furthermore, we use an other block C[hA, wB] to store scores of ungapped alignments between block nb1 and block nb2. The value of each cell[i,j] in block C corresponds to the score of subsequence j (row j of block nb1) and subsequence i (column i of block nb2).

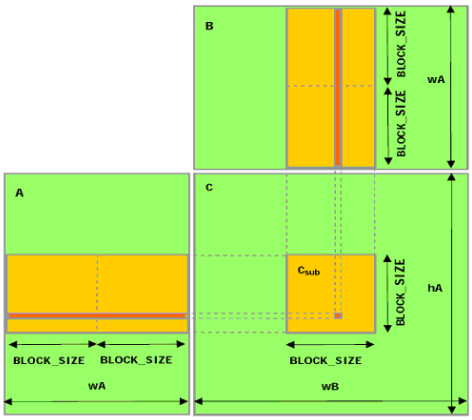

Fig. 2. *Computing ungapped alignment on GPU.*

The task of $|nb1| \times |nb2|$ ungapped alignments between block A and block B is split among threads on GPU as followed: each thread block is responsible for computing on square subblock C_{sub} of C. Each thread within the block is responsible for computing one element of C_{sub} (Figure 2). The dimension block_size of C_{sub} is chosen equal to 16. Thus, there are $\frac{hA}{16} \times \frac{wB}{16}$ thread blocks in grid. Threads within thread block share memory each other.

Two subsequence blocks are mapped to the texture memory of the GPU. The texture memory is shared by all the processors, and speed up comes from the texture memory space by being implemented as a read-only region of device memory. At the beginning of the computation, each thread loads a character from the texture memory to shared memory by a texture reference, called texture fetching.

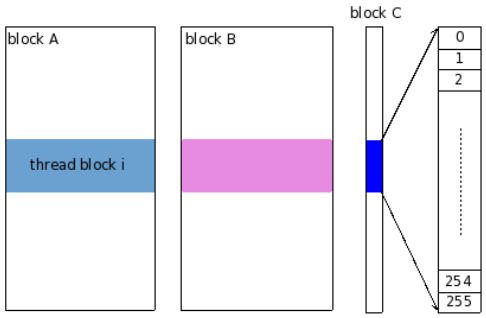

Fig. 3. *Computing small gapped alignment on GPU.*

2) Small gapped alignments: The use of high performance computing on GPU is efficient only to perform large tasks. Thus, we use the GPU to execute small gapped alignments when there are at least K elements ready for computation. With K small gapped alignments, there are 2K extensions extending in two directions. To be enable to compute 2K extensions on the GPU, we have to construct two subsequence blocks as already done for ungapped alignment: one block (A) for bank1 and one block (B) for bank2. Compared to ungapped alignment, there is a difference: for one small gapped alignment, we copy two subsequences in bank1 - one at the left and one at the right of seed - to block A, and two subsequences (bank2) to block B. Consequently, there are 2K subsequences in each block. The GPU divides 2K extensions into thread blocks, each thread within a block is responsible for computing one extension.

The form of thread block is 1×256. Thus, there are $\frac{2K}{256}$ thread blocks. Two subsequence blocks are mapped to the texture memory. At the beginning of the computation, each thread copies its pair of subsequences from the texture memory by the texture reference to its local memory for reducing memory access conflict. The scores of 2K extensions are stored in block C[2K,1] (Figure 3).

IV. RESULTS

We tested the implementations on an Intel 2.6 GHz processor with 2 MB cache L2, and 2 Gb RAM, running Linux (fedora 6). We used the graphic card GeForce 8800 GTX (version GPU). The characteristics of this board are as follows:

- 16 multiprocessors SIMD at 675 MHz; each multiprocessor is composed of eight processors running at twice the clock frequency;
- maximum number of threads per block: 512;
- amount of device memory: 768 MB at 1.8 GHz;
- maximum bandwidth observed between the computer memory and the device memory: 2 GB/s.

The following banks have been used for testing: (1) a protein bank contenting 141,708 sequences from the PIR-Protein bank (the version 80 01/2005) with an average length of 340; (2) a nucleotide bank contenting 27,360 sequences (gbvrt3 in GenBank, the version 156) with an average length of 5,454; (3) a set of four protein banks (extracted from the SWISS-PROT bank, the version 05/2007) contenting respectively 5,000, 10,000, 20,000, and 40,000 sequences with an average length of 367.

The execution time is calculated using the Linux command, "time". The BLASTP and TBLASTN of the BLAST family (version 2.2.16, 2007) were launched with default parameters, except for the statistical parameter, E-value, that is chosen at 10^{-3}, which is a reasonable value in a context of intensive sequence comparison.

The sensitivity is evaluated in relation to the number of alignments found by the two programs. Specifically, we test whether alignments begin and end at the same places in the two banks with a margin of P % calculated on the average size of the two alignments. For example, to compare two alignments of size 100 at 5 %, we check that the start and stop positions of the alignments are in the range of 5 amino acids.

TABLE III compares execution time of (SIMD/GPU) iBLASTP and BLASTP with parameter set as: PIR-Protein bank and four SWISS-PROT (SWP) banks, TABLE IV compares execution time of (SIMD/GPU) iTBLASTN and TBLASTN with parameter set as: GenBank and four SWISS-PROT banks. Acceleration factors of 4.2 and 5.6 are respectively obtained for SIMD iBLASTP and SIMD iTBLASTN.

TABLE III
EXECUTION TIME OF TWO IBLASTP PROGRAMS AND BLASTP

nb. seq. (SWP)	BLASTP	GPU iBLASTP		SIMD iBLASTP	
	(sec)	(sec)	speedup	(sec)	speedup
5k	3,521	640	5.5	857	4.1
10k	6,832	1,186	5.6	1,585	4.3
20k	13,597	2,420	5.6	3,188	4.2
40k	26,111	4,581	5.6	6,053	4.3

TABLE IV
EXECUTION TIME OF TWO ITBLASTN PROGRAMS AND TBLASTN

nb. seq. (SWP)	TBLASTN	GPU iTBLASTN		SIMD iTBLASTN	
	(sec)	(sec)	speedup	(sec)	speedup
5k	7,063	773	9.1	1,328	5.3
10k	13,597	1,373	10.0	2,394	5.6
20k	27,147	2,613	10.4	4,554	5.8
40k	52,232	4,942	10.5	8,554	6.1

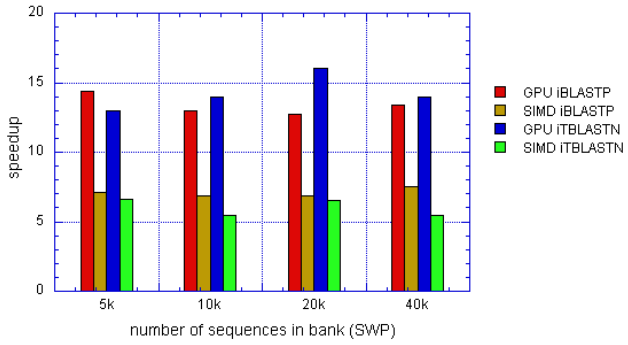

Fig. 4. Speed up of SIMD and GPU for ungapped alignment

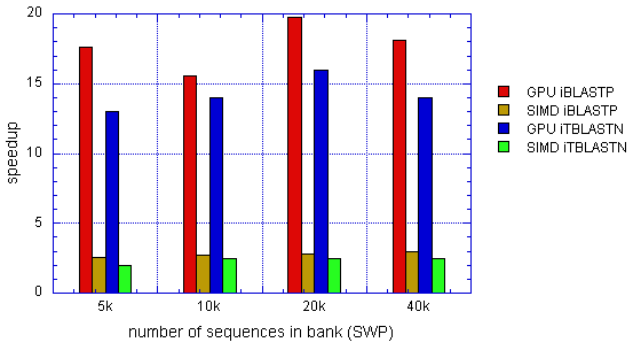

Fig. 5. Speed up of SIMD and GPU for small gapped alignment

TABLE V
COMPARISON OF SENSITIVITY BETWEEN IBLASTP AND BLASTP

nb. seq (SWP)	nb. of alignments		value of P		
	BLASTP	iBLASTP	2%	5%	10%
5k	305,435	305,663	94.7%	94.9%	95.4%
10k	611,031	610,993	95.3%	95.3%	95.8%
20k	1,047,794	1,062,130	94.8%	95.0%	95.4%
40k	2,237,076	2,237,140	94.5%	95.2%	95.6%

TABLE VI
COMPARISON OF SENSITIVITY BETWEEN ITBLASTN AND TBLASTN

nb. seq (SWP)	nb. of alignments		value of P		
	TBLASTN	iTBLASTN	2%	5%	10%
5k	290,016	290,406	95.6%	95.7%	95.9%
10k	572,608	573,880	96.0%	96.2%	96.4%
20k	1,172,466	1,173,378	96.4%	96.5%	96.6%
40k	2,208,330	2,211,935	96.4%	96.6%	96.7%

Acceleration factors of 5.5 and 10 are respectively obtained for GPU iBLASTP and GPU iTBLASTN.

With fine grained parallelism of SIMD instruction, the performance of ungapped extension achieve a speed up ranging from 5 to 7 compared to the standard implementation. Acceleration factors for 13 and 16 are obtained for GPU as reported in Figure 4.

Furthermore, the SIMD small gapped alignment achieves a speed up ranging from 2.5 to 2.8 compared to the standard implementation. Acceleration factors for 13 and 19 are obtained for the GPU as reported in Figure 4.

The main disadvantage of SIMD small gapped alignment is that element scores are sequentially accessed from memory. Consequently, speed up is less than speed up of SIMD ungapped alignment. Compared to SIMD ungapped alignment, the GPU can achieve about 2 times faster. The powerful performance of SIMD ungapped alignment is that it uses effectively the subsequence profile cache and element scores can be accessed in parallel. We believe that the GPU lost performance because of several factors, including poor cache memory and lack of high bandwidth access to cached data.

Based on the definition of two equivalent alignments, we compared the sensibility between iBLASTP and BLASTP (iTBLASTN and TBLASTN). The same data sets as previously are considered. The number of alignments found by the two iBLASTP (iTBLASTN) implementations is the same, thus, we have compared the sensibility between BLASTP and GPU iBLASTP (TBLASTN and GPU iTBLASTN). This sensibility was evaluated by considering three values of P: 2 %, 5 %, and 10 %. TABLES V & VI summarize the results. For each performance, the number of alignments found by BLASTP and GPU iBLASTP (TBLASTN and GPU iTBLASTN) is specified as a percentage of equivalent alignments.

The two programs BLASTP and GPU iBLASTP (TBLASTN and GPU iTBLASTN) detect the same number of alignments, and approximately 95 % (96 %) of the alignments are equivalent. In fact, the difference could be explained as followed: BLASTP and GPU iBLASTP (TBLASTN and GPU iTBLASTN) do not use the same seed. Thus, there are some alignments found by BLASTP (TBLASTN) and not by GPU iBLASTP (GPU iTBLASTN) and inversely.

V. CONCLUSION

In this paper we have presented a parallel subset seed algorithm for similarity search between protein sequences by using SIMD instruction and GPU. To our knowledge, this is the first attempt for this type of algorithm on these structures, the other implementations for this search are based on the dynamic programming algorithm.

Two programs have been developed and tested: iBLASTP and iTBLASTN. They refer to BLASTP and TPLASTN of the BLAST program family that are daily used by thousands

of biologists. For both GPU (SIMD) programs, we have obtained acceleration factors of 5.5 and 10 (4 and 5.6) overall execution time, and gains ranging from 13 to 19 (2.3 to 7) for computation parallelized on GPU (SIMD).

One of the limitations of our current implementation is the sequential treatment of stage 3-2, which limits significantly the overall speed up. The solution may come from works demonstrated in [5] [2]. This stage uses essentially the dynamic programming algorithm which is effective on the GPU and through SIMD instructions. Therefor, we expect to obtain a substantial gain by parallelizing this stage on GPU or using SIMD instruction.

REFERENCES

[1] Benson, D., Karsch-Mizrachi, I., Lipman, D., Ostell, J., .Wheeler, D., GenBank, Nucleic Acids Research, Vol. 35, pp. 21-25, 2007.

[2] Michael Farrar, Striped Smith-Waterman speeds database search six times over other SIMD implementation, Bioinformatics, Vol. 23, no. 2 2007, pp. 156-161.

[3] Peterlongo, P., Noe, L., Lavenier, D., Georges, G., Jacques, J., Kucherov, G., Giraud, M., Protein similarity search with subset seeds on a dedicated reconfigurable hardware, Parallel Bio-Computing, Gdansk, Poland, 2007.

[4] NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.0, 23/6/2007.

[5] Liu, W., Schmidt, B., Voss, G., Schroder, A.& Wolfgang, M., Biosequence database scanning on a GPU, HICOMBO, Rhodes Island, Greece, 2006.

[6] Kucherov, G., Noe, L. & Roytberg, M. A unifying framework for seed sensitivity and its application to subset seeds, JBCB 2006.

[7] Noe, L. & Kucherov YASS, G. Enhancing the sensitivity of DNA similarity search, NAR 2005.

[8] Daniel G. Brown, Optimizing multiple seeds for protein homology search, IEEE/ACM Transactions on Computational biology and bioinformatics, Vol. 2, pp. 29-38, 2005.

[9] Li, M., Ma, B., Kisman, D. & Tromp, J., PatternHunter II: Highly sensitive and fast homology search, Journal of Bioinformatics and Computational Biology, Vol 2(3), pp. 417-439, 2004.

[10] Ma, B., Tromp, J. & Li, M., PatternHunter: faster and more sensitive homology search, Bioinformatics, Vol 18(3), pp. 440-445, 2002.

[11] Zhang, Z., Schaer, A., Miller, W., Madden, T., Lipman, D., Koonin, E., and Altschul, S., Protein sequence similarity searches using patterns as seeds, Nucleic Acids Research, Vol. 26, pp. 3986-9390, 1998.

[12] Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., Lipman, D., Gapped BLAST and PSI-BLAST : A new generation of protein database search programs, Nucleic Acids Research, Vol. 25, No. 17, pp. 3389-3402, 1997.

[13] Altschul, S., Gish W., Miller W., Myers E. W. & Lipman D., Basic Local Alignment Search Tool, J. Mol. Biology, 215, pp. 403-410, 1990.

[14] Wilbur, W. and Lipman, D., Rapid similarity searches of nucleic acid and protein data banks. Proceedings of the National Academy of Sciences USA, Vol. 80(3), pp. 726-730 1983.

[15] Smith, T.F., Waterman, M.S., Identification of common molecular subsequences, J Mol Biol 1981, 147(1), pp. 195-197.