Implementing Protein Seed-Based Comparison Algorithm on the SGI RASC-100 Platform

Van-Hoa Nguyen
IRISA/INRIA Rennes
Rennes, France
vhnguyen@irisa.fr

Alexandre Cornu
IRISA/INRIA Rennes
Rennes, France
cornu@irisa.fr

Dominique Lavenier
ENS Cachan/IRISA
Rennes, France
lavenier@irisa.fr

Abstract

This paper describes a parallel FPGA implementation of a genomic sequence comparison algorithm for finding similarities between a large set of protein sequences and full genomes. Results comparable to the `tblastn` *program from the BLAST family are provided while the computation is improved by a factor 19. The performances are mainly due to the parallelization of a critical code section on the SGI RASC-100 accelerator.*

1. Introduction

Genomic treatments are good candidates for FPGA accelerators since they only need integer computation and small data paths. Data are basically DNA or protein sequences coming from all living organisms from which new knowledge are extracted by comparing intensively their genes and/or genomes.

With the rapid progresses of new biotechnology processes, and especially the next generation sequencing able to generate millions of small DNA sequences in a single run, the bioinformatics discipline is now facing new challenges. The short read sequencing (SRS) technology, for example, opens the door to new possibilities and requires to reconsider basic bioinformatics treatments such as genome and metagenomic annotation, genome resequencing, assembly of closely related species, novo assembly, etc. [9] [11]. All these domains have in common to manipulate an increasing number of genomic sequences (DNA or protein).

This paper focuses on one specific treatment: the comparison of a large set of protein sequences against full genomes. Typically, it is included in bioinformatics workflows for annotating new sequenced genomes. From a set of known proteins, the aim is to locate in the genome regions having significant similarities. Thus, using the genetic code, the genome is first translated into its 6 possible protein frames. As a result, two large sets of protein sequences need to be compared together.

A well known program to perform this task is the `tblastn` program from the NCBI BLAST family [3]. Actually, there are only a few implementations onto FPGA accelerator targeting specifically this program. We can cite the SeqCruncher PCIe board from TimeLogic which implements the Tera-TBLASTN software [1], the Cube from CLC bioinformatics which accelerates a sensitive version called Smith and Waterman tblastn [2], the FPGA/FLASH prototype from IRISA which combines FLASH memory and reconfigurable computing [8], and a systolic approach from NUDT [5].

Implementing the NCBI BLAST programs onto FPGA for comparing two large sets of sequences is not straightforward for the following reasons:

- the BLAST programs have been first designed for scanning purpose: querying large banks (millions of sequences) with a single sequence;
- the internal BLAST algorithm is fundamentally sequential, even if a multithreaded option is available;
- the BLAST programs are highly optimized. It is thus difficult to find new tricks for improving significantly the existing code.

We propose a slightly different way to find similarities between protein sequences. We use the same heuristics as BLAST programs, but the code is structured differently: it considers two large sets of data to process, and not one sequence versus many sequences. As a result, we can reorganize the computation in such a way that the most time consuming part can be localized on a small critical section.

This critical section has been implemented on the SGI RASC-100 platform and performances have been compared with the NCBI `tblastn` program. Speedup values ranging from 6 to 19 has been observed depending on the size of the protein sets.

The rest of the paper is organized as follows: the next section describes briefly our algorithm. Section 3 presents the genomic operator implemented onto the RASC-100 accelerator. Section 4 details the performances and section 5 concludes the paper.

2 Seed-based algorithm

2.1 Algorithm overview

The algorithm is based on a well known and powerful heuristic to detect similarities between two protein sequences. It supposes that two protein sequences sharing sufficient similarities include at least one identical common word of W amino acids. Thus, instead of systematically scanning all sequences, as it is done in the dynamic programming method, only sequences (or part of sequences) with common words are considered. From this common word, extensions on both sides are performed to find larger similarity. These words are called *seeds* since they are the starting point to find regions of interest.

To be efficient, this method supposes first to index the sequences according to words of W characters (amino acids). This index is then used to locate potential seeds from which extensions are performed. Our algorithm follows this idea. More precisely, it is split into 3 distinct steps:

- **step 1:** indexing
- **step 2:** ungapped extension
- **step 3:** gap extension

The first step indexes the sequences of the two banks. If the size of the seed is W, then we construct two W^α entry tables $T0$ and $T1$ (one for each bank) with α equal to the alphabet size (20 for protein). The number of entries reflects the number of possible words of W characters. Each entry k of the table points to an index list (IL_k) of sequence offsets where such a word occurs.

The second step corresponds to the following nested loops:

```
for k = 1 to $W^\alpha$
   for i = 1 to len($IL0_k$)
      for j = 1 to len($IL1_k$)
         ungapped_extension($IL0_k[i]$,$IL1_k[j]$)
```

For all entries of tables $T0$ and $T1$, all elements of the two associated index lists $IL0$ and $IL1$ are considered as potential seeds to start a similarity region. If, for an entry k, $IL0_k$ has K_0 elements and $IK1_k$ has K_1 elements, then there are $K_0 \times K_1$ extensions to proceed. At this stage, the extension procedure is a very simple treatment for deciding if it is worth to start a complete – and expensive – computation. It is named *ungapped_extension* because the similarity computation doesn't consider the possibility to lose or include extra characters (gap) in the solution.

The third step is much more complex. The search space is augmented by the possibility to consider gaps. This operation is triggered only if the neighbouring of a seed (computed in the previous step) presents enough similarity.

2.2 Ungapped extension

Table 1 shows the percentage of time spent in the different steps when comparing 30,000 proteins against the Human chromosome 1. It clearly indicates that the ungapped extension step represents the majority of the execution time. Thus, speeding up the whole algorithm must first target this critical section.

step 1	step 2	step 3
0.3%	97 %	2.7 %

Table 1. Percentage of time spent in the different steps of the algorithm

The ungapped extension procedure aims at rapidly computing a raw similarity to decide if the seed neighbouring has a good probability to generate relevant similarity on a larger region. This is simply done by computing a score depending on the similarity between amino acids. More precisely, a maximum score is computed from the substitution costs between pairs of independent amino acids surrounding the seed.

If we consider two substrings S0 and S1 of length $2 \times N+W$ composed of a seed of W characters with its left and right extensions of N characters, then the maximum score is computed as follows:

```
score = max_score = 0;
for (k=1; k<=2*N+W; k++) {
   score = max(score,
               score + Sub[S0[k]][S1[k]])
   max_score = max(score,max_score);
}
```

where Sub[x][y] is the cost for substituting x by y, and S[k] is the k^{th} character of S. The matrix Sub is generally determined by genomic considerations based on protein evolution theory, for example, the BLOSUM62 matrix [7].

When the score exceeds a given threshold value, the couple (S0,S1) is transmitted to the next stage for further processing (gap extension).

The great advantages of this ungapped extension step are:

- the simplicity of the computation: a score is a sum of small integers;
- the locality of the data: including the score calculation, we have a 4 level nested loops, suggesting a strong reuse of data;
- the regularity: all the substrings have the same length.

These interesting features make this step a good candidate for a parallel implementation on a processor array. The next section describes the architecture of a Parallel Sequence Comparison operator (*PSC operator*) dedicated to protein score computation.

3 Architecture and implementation

3.1 PSC operator architecture

Basically, the PSC operator receives two data flows corresponding to the IL_0 and IL_1 index lists (described in the previous section) and output pairs of integers corresponding to the numbers of the 2 sub-sequences presenting strong similarity.

The architecture has been designed to drive a large number of processing elements (PE). These PEs work in a SIMD fashion and are specialized to compute in parallel the score between one sub-sequence from $IL0$ with several sub-sequences from $IL1$.

A pipeline structure has been chosen for optimizing the clock frequency. Indeed, short and parallel data paths – instead of long and shared data paths – imply shorter delays and makes the Place and Route process easier, especially when the design is using a consequent amount of FPGA resources. Thus, slots (or clusters) of several PEs are separated by registers barriers, which delay and reenforce the signals (data and control) as shown below:

In this architecture, the control is independent of the number of PEs. This is a great advantage for, at least, two reasons: (1) validation and test: a single PE can be used first for simulation, software development, etc. Then, gradually, the number of PEs can be increased to the pipeline length and, finally, set to its maximal size; (2) the design can target different array size depending on the available reconfigurable resources.

The PSC operator architecture is divided into 5 main components:

- *Input Controller 0*: it reads sub-sequences from the $IL0$ port and pushes them into the $IL0$ pipeline;
- *Input Controller 1*: it reads sub-sequences from the $IL1$ port and pushes them into the $IL1$ pipeline;
- *PE Slots*: they are groups of PE with a common result management module made of:
  - *Processing Element (PE)*. One PE stores an $IL0$ sub-sequences. Its main task is to compute a score between the $IL0$ sub-sequence and many $IL1$ sub-sequences;
  - *Result Management Module*. This module scans results from a slot of PE and stores them into a FIFO if the computed score is higher than a threshold value. These FIFOs are cascaded to asynchronously transfer the results to the output port.
- *Output Controller*: it reads data from the cascaded FIFOs and writes them to the Result port;
- *Master controller*: it manages the global architecture: process start, data loading, score computation, results recovering, process end.

3.2 PE architecture

Figure 2 represents the PE architecture. A complete treatment is split into two phases:

- *initialization*. This phase loads an $IL0$ sub-sequence of $W + 2 \times N$ amino acids into the shift register. A feedback loop allows the sub-sequence stored in the shift register to be reused for several computations. The size of the shift register correspond to the size of the sub-sequences ($W + 2 \times N$);
- *computation*. During the computation process, the $IL0$ sub-sequence is sequentially sent, amino acid by amino acid, to the score computation unit together with amino acids coming from the $IL1$ data path.

A computation is performed in $W + 2 \times N$ clock cycles. On each clock cycle, a PE processes one amino acid coming from the $IL0$ sub-sequences and one amino acid coming from the $IL1$ sub-sequence. They are first sent a ROM which output the substitution cost of these 2 amino acids. The result is added to the current score and a maximum value is computed.

After $W+2\times N$ cycles, the maximum is sent to the result management module. It is thus compared with a threshold value. If it is larger, it is sent to the cascaded FIFOs.

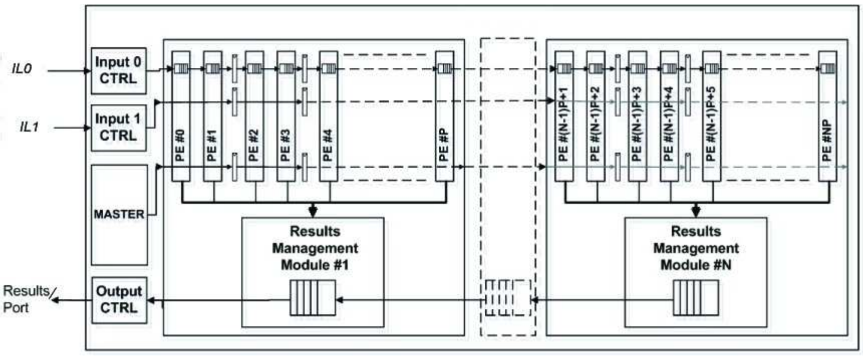

Figure 1. PSC operator architecture

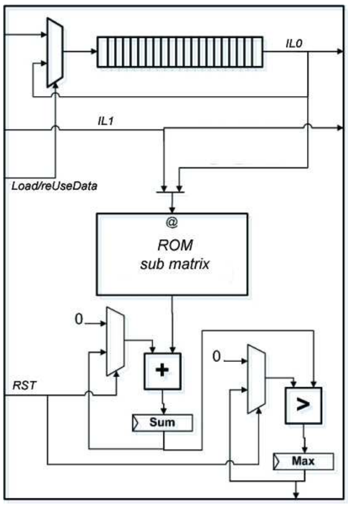

Figure 2. Processing element architecture

3.3 RASC-100 architecture

The PSC operator has been implemented on the RASC-100 (Reconfigurable Application-Specific Computing) accelerator from SGI. This reconfigurable platform is interconnected to the host system through a NUMAlink bus. The RASC-100 is made of two Xilinx Virtex-4 FPGA components, two TIO modules (for connecting the FPGA components to the Altix system), a SRAM memory and a loader module for initializing the FPGA with new bitstreams. In addition, SGI provides a user-configurable interface (SGI Core) for managing DMA transfer, memory access and user registers (Algorithm Defined Registers : ADR)

Figure 3 details the RASC-100 architecture and the way the PSC operator has been integrated in this environment.

4 Performances

We implement our algorithm on the Altix 350 platform composed of an Intel Itanium2 Core2 (1.6 GHz) with 1 MB cache L2, 4 GB RAM, and running SUSE Linux. Steps 1 and 3 are performed on the Altix 350 while step 2 is deported on the RASC-100 accelerator.

Computation time is compared with the NCBI `tblastn` program (Version: 2.2.18) run on the Altix 350 with an E-value set to 10^{-3}, which is a recommended value in the context of intensive sequence comparison. The other parameters are set to their default values. The execution time is calculated using the Linux command, `time`. The following data set has been considered:

- the Human chromosome 1 (220×10^6 nucleotides) translated into its 6 reading frames, comes from Homo sapiens (NCBI Mar. 2008);

- 4 protein banks selected from the non-redundant protein data bank (NCBI Aug. 2008) including respectively 1,000, 3,000, 10,000, and 30,000 proteins and representing respectively 336,232, 1,025,835, 3,433,471 and 10,335,365 amino acids.

4.1 Overall performances

Table 2 gives the execution times for comparing the different protein banks against the Human chromosome 1. It compares the execution times of NCBI BLAST and the RASC implementation with respectively 64, 128 and 192 PEs running at 100 MHz.

Not that this experimentation uses only half of the resources of the ALTIX 350 / RASC-100 platform: only one FPGA is used, and steps 1 and 3 are run sequentially onto one core. To be fair, the NCBI `tblastn` program is also run in a sequential mode (one core). It can be seen that for small protein banks the performance ratio between the RASC implementation and NCBI BLAST ranges from 5 to

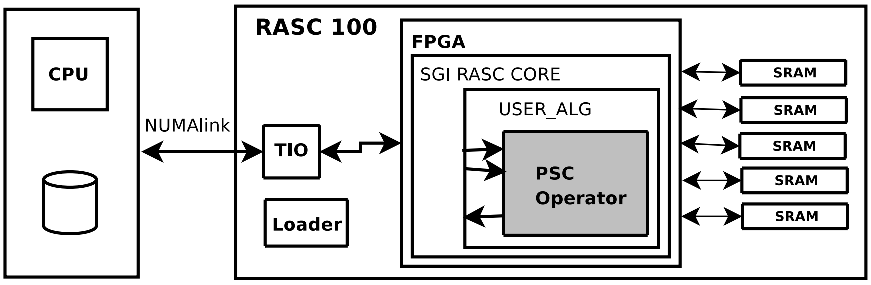

Figure 3. RASC-100 FPGA architecture

Protein bank	1K	3K	10K	30K
1 FPGA	168	223	510	1,373
2 FPGAs	148	175	330	759
Speedup	1.14	1.27	1.54	1.80

Table 3. Performance comparison of 1 FPGA and 2 FPGAs for 192 PEs and the 4 protein banks

10. This is mainly due to the PE array which is not used at its maximal capacity: there are not enough sub-sequences related to one specific seed to feed entirely the array. Another factor is the time for indexing the banks: it remains high compared to the execution time of steps 2 and 3. But, as the bank size becomes larger, the ratio becomes much better.

In addition, a parallel version using the two FPGAs and the two cores has been tested in pthread programming model. But in this version, a synchronization problem between the host system and 2 FPGAs has been encountered while transferring the sequence data and the results. To evaluate performances of 2 FPGAs, we have to modify the condition test by increasing threshold value in ungapped extension procedure. Not that this modification reduces number of ungapped alignments transmitted to the third step, but dose not decrease computing time on FPGA. We only use this condition test for measuring the performance of 2 FPGAs. Table 3 shows performances (in second) of 1 FPGA and 2 FPGAs for 192 PEs. Again, with small protein bank, 2 FPGAs dose'nt achieve good performance. But, as the bank size becomes larger, this performance becomes much better.

4.2 Step 2 analysis

To understand the parallelization efficiency of the ungapped extension implementation, Table 4 reports the time (in second) and speed up measured with and without the RASC-100 accelerator. It clearly depends of the number of PEs and of the size of the data set: larger the amount of data, better the efficiency.

An interesting point to highlight is that the sequential execution time of step 2 of our implementation is slower than the overall execution time of the NCBI tblastn program (first column of the 2 Tables 2 and 4). But this section of code, which represents a very high percentage of time over the total execution time, has been primarily designed to have an optimal efficiency on a parallel support. Hence, executed on the RASC-100 accelerator, the execution time of step 2 is thus strongly divided, leading to significant speedups over optimized sequential implementation such as the NCBI tblastn software.

4.3 Comparison with other tblastn FPGA implementation

One criteria for comparing different implementations can be the amount of data process per second. In the case of the `tblastn` program it is given by the product of the number of *Kilo Amino Acids (Kaa)* and the number of *Mega nucleotides (Mnt)* divided by the processing time. Based on this ratio, Table 5 compares various implementations.

DeCypher engine has been benchmarked[1] for comparing 4289 proteins (1,358,990 aa) against 192 bacterial genomes (775,191,168 aa) in 1 hour and 36 minutes. The next version, called SeqCrunch, provides probably better performance, but no data are available. For CLC, values are extrapolated from [2] where performance are given in GCUPS (Giga Comparison per second) which are similar to our measure. The comparison is strongly biased since the CLC implementation is very sensitive and based on the dynamic programming method. The FLASH/FPGA board, also developed in our team, provides similar results but requires specific hardware not available on the market [8]. The performance of the Systolic approach given in [5] are peak performance measured on a FPGA prototype when the protein sequence length exactly match the size of the array (3072 PE). A standard protein (330 aa) search gives an average

[1] http://www.timelogic.com/benchmark_blast.html

	NCBI tblastn	RASC 64 PE	Speedup	RASC 128 PE	Speedup	RASC 192 PE	Speedup
1K protein	2,379	506	4.70	451	5.27	443	5.37
3K protein	7,089	873	8.10	689	10.20	631	11.23
10K protein	24,017	2,220	10.81	1,661	14.45	1,450	16.56
30K protein	70,891	6,031	11.75	4,312	16.44	3,667	19.33

Table 2. Performance comparison of NCBI BLAST and our FPGA implementation. Time is given in second.

	Sequential	RASC 64 PE	Speedup	RASC 128 PE	Speedup	RASC 192 PE	Speedup
1K protein	2,368	220	10.76	176	13.45	169	14.01
3K protein	7,577	462	16.40	280	27.06	223	33.97
10K protein	24,687	1,366	18.07	720	34.28	510	48.38
30K protein	73,492	3,932	18.68	2,015	36.47	1,373	53.52

Table 4. Performance comparison of step 2 only for the 3 different sizes of PE array and the 4 protein banks

	FPGA-RASC	NBCI-BLAST
ROC_{50}	0.468	0.479
AP-Mean	0.447	0.441

Table 6. ROC_{50} and AP-Mean scores of RASC and NCBI BLAST

ratio of 258. In addition, the performance of Systolic implementation dose'nt include gap extension stage.

4.4 Sensitivity and selectivity

Another important issue is the quality of the results produced by the RASC-100 implementation. We use the same seed heuristics as the BLAST algorithm but in a rather slightly different way. In the NCBI BLAST algorithm, the ungapped extension is started when two seeds of 3 amino acids are detected in a closed neighbouring. In our implementation we consider only one seed of 4 amino acids, but based on the subset seed approach [10]. The main reason is that this approach is very efficient for indexing the protein sequences. Theoretically, both approaches have the same sensitivity.

Table 6 reports the receiver operating characteristic (ROC_{50}) and the average precision (*AP-Mean*) scores of both RASC and NCBI BLAST, in which all parameters are set to their default values. The ROC curves and AP-Mean were generated by analyzing the results of aligning 102 queries against the yeast genome in [6]. They correspond to standard procedures to evaluate sensitivity and selectivity of sequence comparison algorithms. Similar values indicate similar sensitivity and selectivity.

More precisely, the ROC_{50} value is calculated as follows: each protein sequence is compared with the yeast genome and the first 100 best hits are marked as true or false positives according to a careful human expert annotation. True positives are sequences of the same family. Then, for each of the first 50 false positives, the number of true positives with a higher score is get. These numbers are added and the sum is divided by $50 \times P$, P being the number of sequences of the family. The average of these ROC_{50} scores gives the final ROC_{50} score.

The average precision (AP) criterion is borrowed from information retrieval research as described in [4]. For calculating the average precision, the 50 best alignments per query are marked as either true or false positives. For each true positive found by the comparison algorithm, the true positive rank is divided by its position. All these numbers are summed up and divided by the total number of true positives, giving one AP value per query. The Mean-AP is the average of all the APs.

5 Conclusion

We have proposed an FPGA implementation of the `tblastn` algorithm on the SGI RASC-100 accelerator. Compared to the NCBI BLAST software version, the RASC-100 implementation provides a speed up of 19 for processing large amount of data. This has been achieved by rewriting the code in a way suitable for parallel processing on hardware accelerators.

The heart of the hardware architecture is based on the parallel score calculation between two short amino acid sequences. In that way, this design can be directly reused for implementing `blastp`, `blastx`, and `tblastx` BLAST

DeCypher	CLC	FLASH/FPGA	Systolic	1/2 RASC-100
182	2	451	863	620

Table 5. Number of Kilo amino acids x Mega nucleotides processed par second (*KaaMnt/sec*)

Protein bank	1K	3K	10K	30K
step 1	43 %	31 %	14 %	6 %
step 2	38 %	35 %	35 %	37 %
step 3	19 %	34 %	51 %	57 %

Table 7. Percentage of time spent in the different step of RASC with 192 PEs for 4 protein banks

family programs.

Improving the parallelization of step 2 (*ungapped_extension*) would provide better overall performances, but would be limited by the execution time of step 3 as shown on the code profiling, Table 7, when using the RASC-100 with 192 PEs.

Now, step 3 has the largest execution time. Hence, optimizing global performances implies now to consider a larger array with faster PEs for ungapped extension together with the design of another reconfigurable operator dedicated to the computation of similarities including gap penalty. The RASC-100 architecture would perfectly support this double activity since it allows two different designs to run concurrently on its two FPGAs.

Also, another way to further optimize would be to consider the next processor generation which will include 4, 8 or more cores. As a matter of fact, when such processors will be linked to reconfigurable resources, the question will be how to dispatch the overall computation between cores and FPGA to get optimal performances. The next platforms involved in reconfigurable super computing will have to deal with this matter to find the best compromise and to decide which part of the application will be more suited for reconfigurable implementation.

References

[1] Timelogic seqcruncher^{TM} PCIe accelerator card, http://www.timelogic.com/seqcruncher.html.

[2] White paper on CLC bioinformatics cube 1.03, clc bio, 2007, http://www.clcbio.com.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

[4] Z. Chean. Assessing sequence comparison methods with the average precision criterion. *Bioinformatics*, 19:2456–2460, 2003.

[5] X. Fei, D. Yong, and X. Jinbo. Fpga-based accelerators for blast families with multi-seeds detection and parallel extension. In *Bioinformatics and Biomedical Engineering in The 2nd International Conference*, pages 58–62, 2008.

[6] M. Gertz, Y. K. Yu, R. Agarwala, A. Schaffer, and S. Altschul. Composition-based statistics and translated nucleotide searches: Improving the tblastn module of blast. *BMC Biology*, 2006.

[7] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89(22):10915–10919, 1992.

[8] D. Lavenier, G. Georges, and X. Liu. A reconfigurable index flash memory tailored to seed-based genomic sequence somparison algorithms. *VLSI Signal Processing*, 48(3):255–269, 2007.

[9] E. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133–141, 2008.

[10] P. Peterlongo, L. Noe, D. Lavenier, G. Georges, J. Jacques, G. Kucherov, and M. Giraud. Protein similarity search with subset seeds on a dedicated reconfigurable hardware. In *Parallel Bio-Computing (PBC-07)*.

[11] M. Pop and S. L. Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, 24(3):142–149, 2008.