HAL
archives-ouvertes.fr

Seed-based Genomic Sequence Comparison using a FPGA/FLASH Accelerator

Dominique Lavenier, Liu Xinchun, Gilles Georges

To cite this version:

Dominique Lavenier, Liu Xinchun, Gilles Georges. Seed-based Genomic Sequence Comparison using a FPGA/FLASH Accelerator. International IEEE Conference on Field Programmable Technology, Dec 2006, Thailand. http://fpt.selfip.org/fpt06/program.php. hal-00179994

HAL Id: hal-00179994
https://hal.archives-ouvertes.fr/hal-00179994

Submitted on 17 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Seed-based Genomic Sequence Comparison using a FPGA/FLASH Accelerator

Dominique Lavenier[1,3]
IRISA / CNRS
Rennes, France
Email: Dominique.Lavenier@irisa.fr

Liu Xinchun[2,3]
Institute of Computing Technology
Beijing, China
Email: lxc@ncic.ac.cn

Gilles Georges[1,3]
IRISA / INRIA
Rennes, France
Email: Gilles.Georges@irisa.fr

***Abstract*— This paper presents a parallel architecture for computing genomic sequence alignments using seed-based algorithms. Originality comes from the simultaneous use of FPGA components and FLASH memories. The FPGA technology brings the computer power while the FLASH memory provides high memory bandwidth able to feed a large array of specific operators. A 64 GBytes FLASH memory connected to a Xilinx Virtex-2 Pro PCI board has been developed and an array of 160 distance-computation operators have been implemented to perform the first step of seed-based alignment algorithms. Compared to the BLAST reference software family, we measured a speed-up of 75 on a real intensive genomic sequence comparison application.**

I. Introduction

For the last ten years, biotechnology improvements in the sequencing area led to generate an increasing amount of genomic data. More precisely, the size of the genomic databases are doubling every 16 months [2]. This exponential growth provides a priceless source of knowledge which can be advantageously exploited by *in-silico* studies. Today, wet lab experimentations are systematically preceded by a deep analysis of recent data available in the public databases. Results obtained from these analyses limit the number of *in-vivo* experimentations, saving both time and money.

On the other hand, processing large volume of genomic data may require huge computational resources. Genomic sequence comparison programs are traditionally used for that purpose, particularly programs from the BLAST package [13]. Typically, a set of genomic sequences is compared to a specific database or an entire genome. The result is a list of alignments giving a measure of similarity between genomic sequences. Detecting similarities through alignments at the text level often gives clues for determining relationship about gene functionalities.

An alignment is expressed as follows:

```
ATTAGGACCAGGGGATATGA...GACCCAGGAATTTA.GGGA
||| |||| ||||   |||||    ||||| || ||||| ||||
ATTCGGAC.AGGGATTATGAGGAGACCCGGGGATTTAGGGGA
```


[1]This work is supported by the French Research Ministry under the ACI Masses de Données ReMIX

[2]This work is supported by the National Natural Science Foundation of China (Grant No.60573163)

[3]This work is supported by the French-Chinese Research Program PRA SI04-04

It is composed of two substrings. Their size may differ since gap errors (dot characters) can be inserted. A score is calculated according to the number of errors found (substitution or gap). It is linked to an expected value reflecting the biological significance of the alignment.

Many algorithms have been developed to compute alignments. The historical one is due to Wagner and Fisher [17] for simply calculating a distance between two strings of characters. It has been extended by Smith and Waterman [16] to find the best local alignment between two biological sequences and, later on, improved by Gotoh [15] to better reflect biological reality. All these algorithms use dynamic programming algorithms. Because of their high complexity, they are not suited for intensive comparisons. Since the end of the 80's, to face the genomic data explosion, seed-based heuristics have been proposed [14] [13]. They do not guarantee to find the best alignments but, in practice, they produce very satisfying results. The great advantage is that the computation time is drastically reduced.

The BLAST [9] family programs belongs to this class of algorithms. This is today the most popular software both for scanning large databases and for comparing large genomic data sets. Even if this program is fast, handling genomic sequences of billions of nucleotides may take a very long time. Accelerating this kind of algorithm is still worthwhile.

Whereas many efforts have been done to implement dynamic programming algorithms on dedicated parallel architectures [6] [10] [11] [12], seed-based implementation mainly remains a software approach. Today, as far as we know, the only available commercial board accelerating the BLAST program family has been developed by TimeLogic Inc. through the DeCypher FPGA Engine [18]. And it is not clear how BLAST is implemented. Academic research includes work on the Mercury system [5] [3] and on the RDISK system [7] [8]. Both projects consider the database as a data stream and perform fast hardware filtering to select regions of interest for further processing. Another interesting idea (RC-BLAST) is based on an open source hardware implementation to only speed-up some critical parts of the BLAST software [1]. Unfortunately, this direct approach doesn't provide significant speed-up, but gives good clues towards advanced hardware implementations.

The approach we propose here is new in the sense that we don't scan the database. Instead, the database is formatted as an index structure from which regions of interest can be directly accessed. The index is built using-BLAST similar seed-based heuristics. This mechanism provides a very efficient way to speed-up the first steps of the genomic sequence comparison algorithms as long as the index memory is able to sustain a fast data access and a high bandwidth. Unfortunately, the size of the database index can be very large (100 GBytes), which makes the use of a simple DRAM memory impossible. In this paper, we show how a large capacity FLASH memory tightly connected to FPGA resources can best implement the first steps of seed-based algorithms.

For the problem introduced above, a PCI-based system has been designed. It includes a 64 GBytes FLASH memory and a Virtex-2 Pro FPGA component. It has been successfully tested on a real and time-consuming application: the comparison of 400,000 proteins against the Human genome. We measured a speed-up of 75 compared to a stand alone 3 GHz PC running the TBLASTN program.

The rest of the paper is organized as follows. The next section presents one of the BLAST program family, TBLASTN, which is used as a reference for the process of comparing proteins versus DNA databases. Section 3 details the database index structure used in our hardware implementation. Section 4 describes the parallel architecture implemented on the reconfigurable resources. Section 5 deals with implementation and result issues. Section 6 concludes this paper.

II. TBLASTN ALGORITHM

The TBLASTN program belongs to the BLAST program family [9]. It aims to find alignments between a query protein sequence and a nucleic database. The result is displayed as protein-based alignments, requiring first the database to be translated into protein sequences before the comparison process. The (simplified) TBLASTN algorithm can be described as follows:

```
index the query with 3-AA words
  for all DNA sequences of the database
    translation into 6 proteins (P1 to P6)
      for all Pi (0<i<7)
        for all 3-AA words
          if word exists in the query index
          then
            hit found
            compute ungapped alignment
            if score > threshold_1
            then
              compute gapped alignment
              if score > threshold_2
              then
                display alignment
```

The first step consists in indexing the query protein sequence: for each 3-amino acid word (3-AA word), a list of positions is attached. The list gives the position (the index) of the words where they appear in the sequence. This data structure is called the *query index*. For example, suppose the following (small) protein sequence:

```
T  P  A  T  Y  K  L  P  E  W  G  G  E  S
1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

The index associated to this protein is:

```
ATY   3        PAT   2
EWG   9        PEW   8
GES   12       TPA   1
GGE   11       TYK   4
KLP   6        WGG   10
LPE   7        YKL   5
```

Once the query sequence is indexed, the next step is to translate all the DNA sequences of the database into the 6 possible potential proteins. Indeed, a DNA sequence can be translated starting from any nucleotides following either the direct or reverse DNA strand. As the genetic code translates an amino acid from a 3-uplet nucleotide (codons), there are 3 encoding possibilities by frame, leading to 6 possibilities if the direct and reverse strand are considered. Suppose the following DNA sequence is given:

```
ACGCAAGCTACCTATAAATTACCTGACTGGGGAGAGTCAACA
```

The reverse strand is given by:

```
TGTTGACTCTCCCCAGTCAGGTAATTTATAGGTAGCTTGCGT
```

The sequence is first inverted and the nucleotides are replaced following the (A $\leftrightarrow$ T) and (C $\leftrightarrow$ G) rules. The first ACG... characters become the last ...CGT characters. From these 2 strands, using the genetic code, we get the 6 following potential proteins:

```
  ACGCAAGCTACCTATAAATTACCTGACTGGGGAGAGTCAACA
1 T  Q  A  T  Y  K  L  P  D  W  G  E  S  T
2  R  K  L  P  I  N  Y  L  T  G  E  S  Q
3   A  S  Y  L  X  I  T  X  L  G  R  V  N

  TGTTGACTCTCCCCAGTCAGGTAATTTATAGGTAGCTTGCGT
4 C  X  L  S  P  V  R  X  F  I  G  S  L  R
5  V  D  S  P  Q  C  G  N  L  X  V  A  C
6   L  T  L  P  S  Q  V  I  Y  R  X  L  A
```

At this point, protein-protein comparison can now be considered. Alignments are thus searched using seed-based protein heuristics. The hypothesis is that the two subsequences which compose an alignment share, at least, one identical 3-AA word. Thus, instead of considering all the possible alignments, we only focus on the alignments having this property. Practically, this technic finds most of the significant alignments while decreasing the computation time from 2 or 3 orders of magnitude as compared to exact methods such as dynamic programming methods.

The next step is to detect the 3-AA common words between the query sequence and the six translated sequences. This is rapidly performed using the query index computed at step 1. When a hit is found (that is two identical 3-AA words), an ungapped alignment is calculated by extending the comparison on its left and right hand side. This treatment doesn't consider gap error. A score is simply calculated by adding substitution

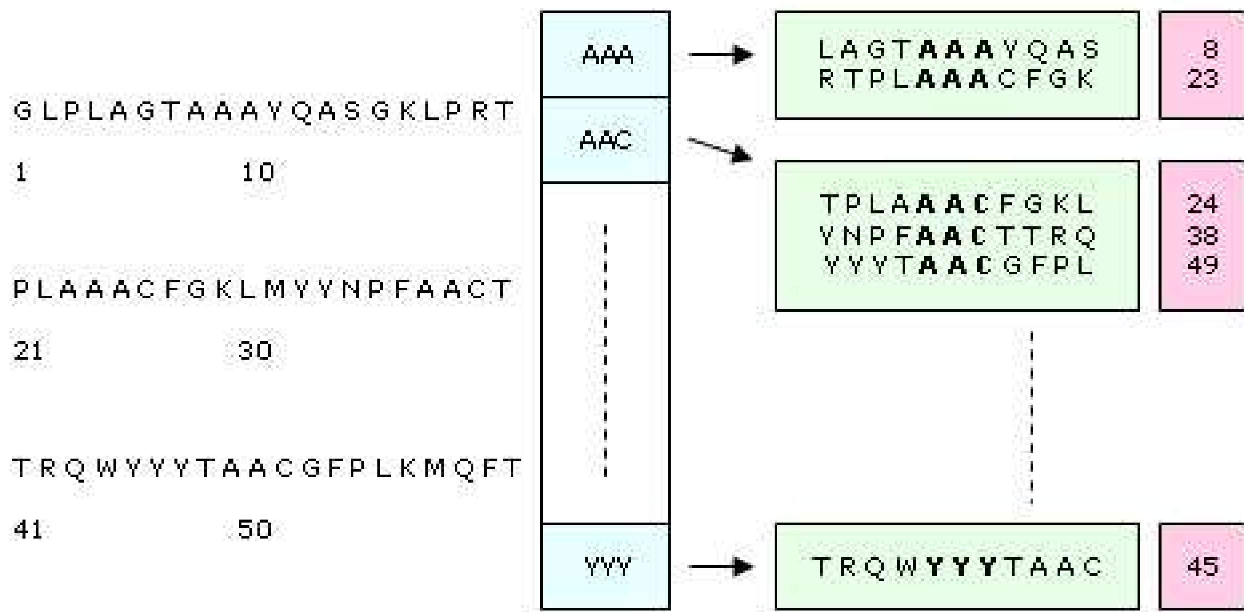

Fig. 1. Structure of the index. All the 3-AA words of the sequences are grouped together. Then, each 3-AA word is associated with its position in the sequence and its neighboring environment. For example, the 3-AA word AAC is present at position 24, 38 and 49. This entry is thus linked to 3 amino acid substrings (TPAAACFGKL, YNPFAACTTRQ, YYYTAACGFPL) which will be used to start the computation of ungapped alignment.

costs between pairwise amino acids. Suppose that the KLP word is processed: it is present both in the query (at position 6) and in the frame #1 and #2 of the database. The 2 following hits can be set:

```
Query      T P A T Y K L P E W G G E S
           |   | | | | | |   | |
Frame #1   T Q A T Y K L P D W G E S T

Query      T P A T Y K L P E W G G E S
                     | | |
Frame #2           R K L P I N Y L T G E S Q
```

Clearly, the first hit position is better and lead to a higher score when an extension is made. The second one will be discarded. This ungapped alignment step can be seen as a filter step to select *promising* hits, that is, hits with a favorable environment likely to produce significant alignments. The selection is based on the score value: only extended hits with scores higher than a threshold value are transmitted to the final step.

The last step computes alignments with gap errors. The hit acts as an anchor from which the final alignment is found. In our example we can still increase the length of the first ungapped alignment by adding one gap as follows:

```
Query      T P A T Y K L P E W G G E S
           |   | | | | | |   | |   | |
Frame #1   T Q A T Y K L P D W G . E S T
```

The principle of the algorithm can be summarized as a 3 step procedure:

- step 1: find hits
- step 2: extend hit (compute ungapped alignments)
- step 3: process alignment (compute gap alignments)

Previous studies have shown that most of the computation time is spent in steps 1 and 2. The following table, extracted from [5], precisely indicates, for various query length, the percentage of time spent in stage 1, 2 and 3.

Query size	word matching (step 1)	Ungapped extension (step 2)	Gapped extension (step 3)
10K	86.50 %	13.30 %	0.20 %
100K	83.30 %	16.60 %	0.10 %
1000K	85.30 %	14.65 %	0.05 %

Obviously, to get a significant speed up, both step 1 and 2 need to be accelerated. Before presenting how an hardware accelerator can handle these 2 steps, we first introduce a suitable data structure allowing fast data accesses and able to rapidly feed hardwire operators.

III. DATABASE INDEXING

In the context of intensive genomics computation, a large set of query proteins (let say N proteins) are compared to a DNA database. In this case, the database is processed N times. From a computational point of view, this approach is quite inefficient since the database need to be systematically re-translated into the 6 frames. Actually, the TBLASTN software

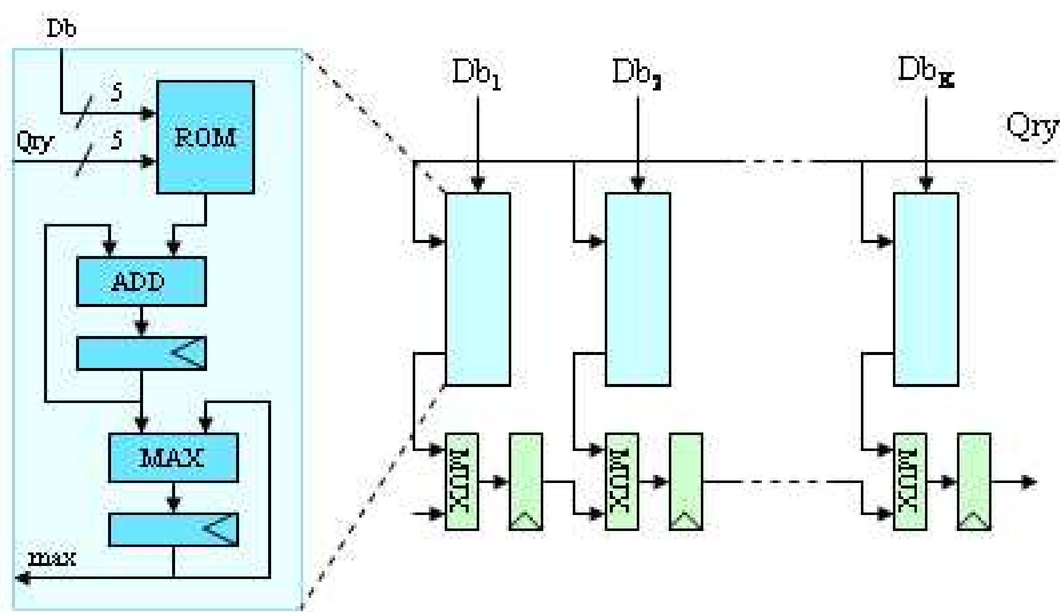

Fig. 2. Ungapped Operator. Left side: architecture principle of an ungapped operator. The substitution cost of the amino acids coming from the query and databse substrings are added. The return score is the maximum value achieved during the calculation. Righ side: K ungapped operators can be connected in such a way that one query substring is compared with K database substrings. Extra hardware are added for collecting the results by shifting the maximum values calculated during the previous cycles.

has an optimizing parameter allowing the user to concatenate a few query protein sequences into a single one. Setting this optional parameter greatly improves the efficiency, but doesn't remove the need to periodically recomputing the 6 frames.

To avoid this drawback, the database can be permanently encoded as a 6 protein sequence set resulting from the 6-frame DNA translation. Consequently, compared to the BLAST approach, the whole process can be inverted: instead of indexing the N queries and reading N times the database, the database is fully indexed and the N queries are sequentially processed, word by word.

Furthermore, to optimize the database access, the index structure is enhanced with extra information: a 3-AA word is not only linked with a list of positions but also with neighboring knowledge. The goal is to be able to rapidly decide if a 3-AA hit has a favorable environment to start an alignment. Thus, each 3-AA word in the index is tied with a neighboring amino acid substring representing its environment (see figure 1). This information allow short ungapped alignments to be immediately computed, avoiding millions of random accesses to the database.

To illustrate the database index structure, consider again the DNA sequence of the previous example and its 6 frame translations. The KLP word of the database will be indexed as follows:

```
KLP --> ATY DWG #1 16
        **R INY #2 5
```

It is present in frame #1 at position 16 with a left neighboring environment ATY and a right neighboring environment DWG. It is also present in frame #2 at position 5 with a (**R, INY) environment.

When a KLP word is found in the query sequence, all the KLP words of the database are immediately accessed together with their neighboring environment. The great advantage is that we only focus on the useful parts of the database. A systematic scan is no longer required.

On the other hand, the size of the index becomes very large: for all words of the six frames, in addition to their positions, two short substrings reflecting the neighboring amino acid environment need to be memorized. As an example, storing a 40 amino acid substring environment leads to a 150 GBytes index for the Human genome. This is 50 times more than the raw data. The database index structure is represented figure 1. Using this structure, and supposing that the database index already exists, the algorithm becomes:

```
 1  Input query
 2  For all 3-AA words W in the query
 3   Go to the environment list linked with W
 4   For all environment
 5    Compute ungapped alignment
 6    If score  > Threshold_1
 7     Then
 8      Compute gapped alignment
 9      If score  > threshold_2
10       Then
11        Display alignment
```

Compared to the TBLASTN algorithm, the finding-hit step is suppressed since statistically, when indexing a large database (such as a complete genome), all the possible 3 amino acid words exist and are associated to a list of positions. Thus,

we can directly process the ungapped alignments from the list associated to every 3-AA words. Accelerating this algorithm means to focus on lines 3 to 6. The next section describes the hardware architecture to perform this calculation.

IV. Architecture

The critical part of the algorithm is to compute a large number of ungapped alignments from two short amino acid subsequences. Let L be the length of the protein subsequences, Qry a subsequence from the query and Db a subsequence from the database. Computing the best score based on an ungapped alignment is done as follows:

```
max = 0;
score = 0;
for (i=0; i<L; i++) {
  score = score + SubMat(Qry[i]][Db[i]);
  if (score>max) max = score;
}
```

The SubMat function returns a positive or negative value according to the substitution cost of amino acid Qry[i] by amino acid Dd[i].

Hardwiring this operator is straightforward as depicted figure 2 (left). This is a serial operator which need L cycles (the length of the substrings) to compute a maximum score value. The substrings Qry and Db are supposed to be sent character by character, one every cycle. The substitution cost is memorized into a ROM memory directly addressed by the amino acids code. As there are 20 different amino acids, they are 5-bit encoded.

One query substring need to be compared with a large number of database substrings. These comparisons are independant and can be performed in parallel. If we suppose that K database substrings can be accessed together, a parallel architecture made of K ungapped operators can be simply implemented. Figure 2 (right) shows the interconnection of K operators.

The Qry substring is broadcasted to all the ungapped operators while K database substrings are connected to K different operators. The system is synchronous: all computation start at the same time, and after L cycles K new results are available. Extra hardware has been added for collecting the results: at the end of the L cycles, they are stored in a register and during the K next cycles they are sequentially shifted to the output. This shift mechanism works only when L (number of cycles to compute a score) is higher than K (the number of operators).

In that scheme, to reach a maximal efficiency, the number of operators depends on:

- the available bandwidth (B) between the database storage support and the reconfigurable resources;
- the clock frequency (F);
- the input data width (dw). Here amino acids are 5-bit encoded (dw=5).

Consequently, the number of operators can be determined as the following ratio:

$$\#operators = \frac{B}{dw \times F} \tag{1}$$

Remember that the size of the index databases is important. It can range from 30 to 300 GBytes, or even more. This is an easy fit to today's disk capacity. However, limitations come from both the time for accessing the data (a few milliseconds) and the limited bandwidth. As an example, with a 50 Mo/sec bandwidth and a 40 MHz clock frequency, it is impossible to implement more than 2 operators.

Using Random Access Memory appears as an attractive alternative to sustain a higher bandwidth, but the complexity of managing hundreds of GBytes of DRAM, the power consumption and the volatility of the data, seriously compromise the use of this technology.

A better solution is FLASH memory technology. It is faster than magnetic storages in terms of data access and bandwidth, it has a low power consumption compared to DRAM and it permanently stores the data. Drawback are a much higher price (compared to disk), a slower data access (compared to DRAM) and a limited write cycles.

To investigate this approach, we have designed a 64 GBytes FLASH memory allowing both to store large database index and to support a bandwidth of 640 MB/sec. Using this memory, 25 ungapped operators can be fed simultaneously. Actually, this represents a small amount of reconfigurable resources available in the Xilinx XC2VP30 FPGA component. There are still resources available to fit many more operators.

In the context of comparing a large set of proteins against large DNA sequences (chromosomes), identical 3-AA words coming from the proteins can be gathered and processed simultaneously. Thus, a few parallel ungapped operators working on different data from the query proteins, but on the same data from the database can be advantageously implemented in order to fill the FPGA and to reach a higher efficiency. The resulting architecture is shown figure 3.

This is an array of $P \times K$ ungapped operators. This array is steered by a controller whose task is to feed the array with query data and to collect the results from the ungapped operators. To reduce the output data flow, results from the ungapped operator are filtered: only the values higher than a threshold value are transmitted.

A basic treatment consists in comparing P query substrings having in common the same 3-AA hit word. The PowerPC processor available in the Virtex-II Pro component controls the treatment. It receives the P substrings from the PCI interface together with information to locate, in the FLASH memory, the 3-AA database word list. Then it runs the whole computation and transfers back the results to the PCI interface. The results which are returned correspond to the locations where significant ungapped hits have been found together with the ID of the matching query.

V. Implementation and Results

A. Platform

An array of 8×20 ungapped operators have been implemented and successfully tested. The platform is a commercial board from Avnet (ADS-XLX-V2PRO-DEVP30). It includes a Xilinx Virtex-2 Pro (XC2VP30), 128 MBytes of DDR

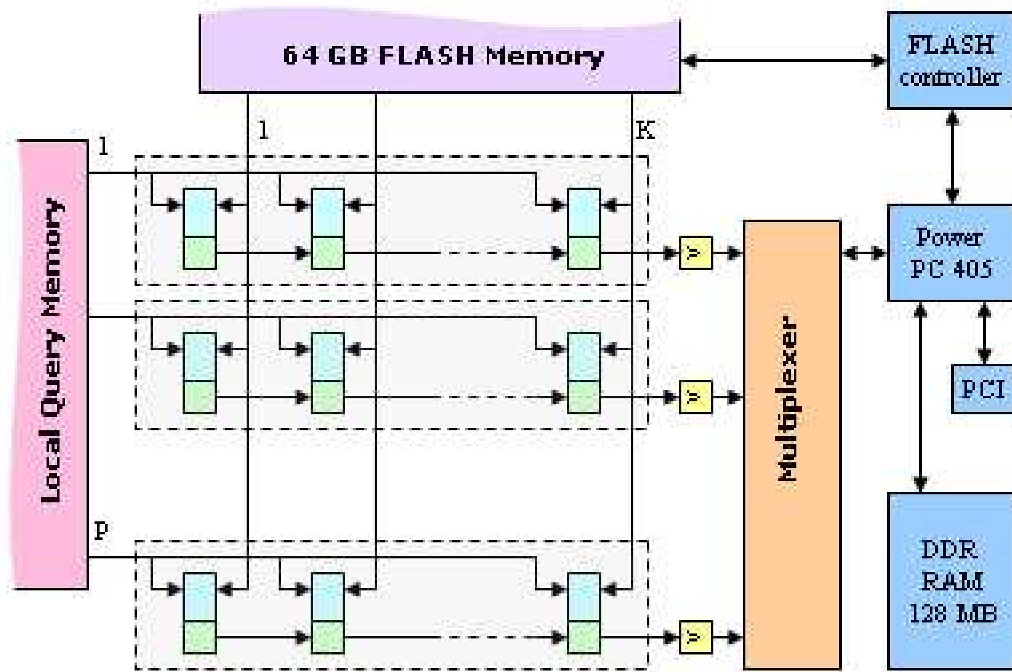

Fig. 3. Architecture of the accelerator. An array of P × K ungapped operators is both feed from the FLASH memory and from a small local memory implemented with the FPGA RAM resources. The local memory contains K substrings which are re-initialized by the PowerPC processor each time a new computation is done.

SDRAM, a PCI bridge and 4 connectors to plug daughter boards.

In addition, we have developed a FLASH memory board of 64 GBytes which can be connected to the Avnet board. 64 1-GB NAND-FLASH chips are interconnected into 4 independent 16 GB banks. Each bank has its own controller implemented into specific Xilinx Spartan-3 FPGA (see figure 4). The maximal bandwidth is 640 MB/sec: each bank is able to deliver a 32-bit data every 25 ns.

NAND-FLASH memory cannot be used efficiently as pure random access memory. To be efficient, data need to be read by pages. In our case, this is not a major problem since we have to read a list of 3-AA word environments which may represent a few consecutive pages. However, access times need to be considered. Typically, it takes about 20 μsec to get a block of data before being able to process it. This latency may highly penalize performance by including idle period each time a new block of data is accessed. The FLASH controllers we designed have integrated advanced mechanisms for anticipating the read of new blocks: while data from a page are output, a new page can be concurrently accessed. This *overlap* mechanism allows high bandwidth to be sustained if the data access can be predicted. Once again, in our case, the algorithm fits to these requirements: all the data accesses (i.e. all the lists of all 3-AA words of the query) can be progressively forecasted as the query input is processed. Sending periodically a list of reading commands to the FLASH controller reduces significantly the access time overhead. In the best case, we only pay the first page access, the other ones are *free*.

B. Application

The FPGA/FLASH architecture have been tested on a real, intensive comparison genomic application. The scope of this paper is not to precisely describe the biological context, but to give an idea of the volume of data to process at an initial stage of a large scale genomic project. The study aims to detect new mitochondrial proteins in the Human genome. As mitochondria may originate from ancestral bacteria, a systematic comparison with the proteome of all available bacteria is done. From a computational point of view, about 400 000 proteins have to be compared with all the chromosomes of the Human genome.

Usually, the TBLASTN program is used. Experiments have shown that performing this computation on a 3 GHz processor will require about 20 000 hours (or 27 months) of computation. With a 64-node cluster, supposing no communication overhead, it will approximately take 13 days.

A program similar to TBLASTN has been developed for integrating the hardware version of step 1 and 2 in our FPGA/FLASH architecture. As the FLASH memory cannot hold the full index of the Human genome, we process each chromosome sequentially. The size of the largest Human chromosome is 246×10^6 nt, leading to an index of nearly 12 GBytes. This easily fit into the FLASH memory. Hence, the whole process is done as follows:

```
for each chromosome:
  - download chromosome index into FLASH
  - process 400,000 proteins
```

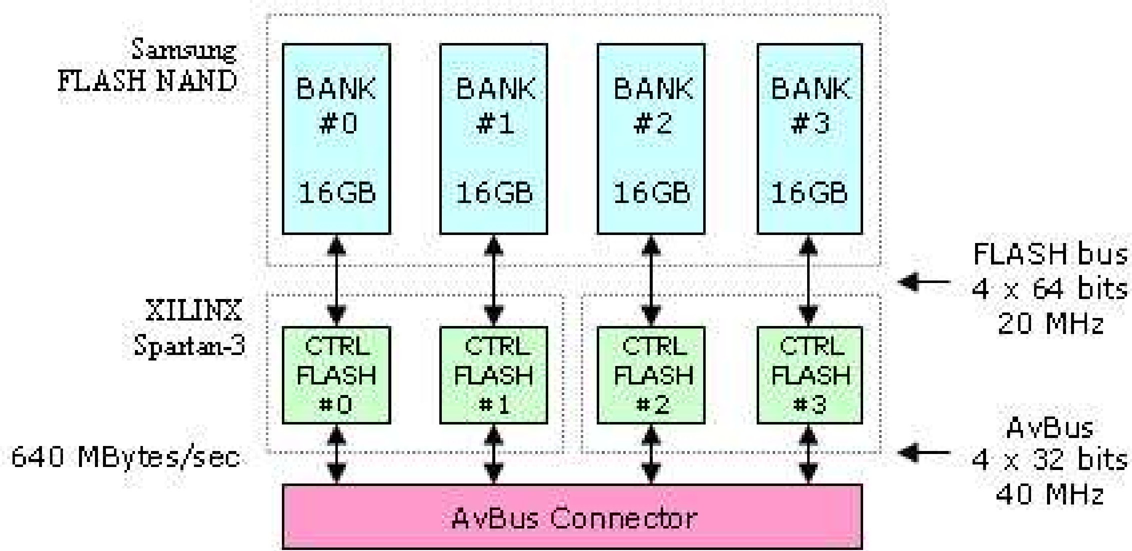

Fig. 4. FLASH memory organization. It is composed of 4 independant 16 GBytes banks. Each bank has its own controller implemented into Xilinx Spartan-3 FPGA. These controllers permit page reading and page access overlaping, leading to sustain a 640 Mbytes/sec bandwidth.

C. Performances

The complete computation has been performed in approximately 260 hours (11 days). It includes the time for downloading new data into the FLASH memory, but not the time for calculating the index of each chromosomes. We consider that this operation need to be done once and that the resulting data index structure can be re-used for many other applications, as it is done for the BLAST family programs.

This experiment shows that connecting a PCI reconfigurable board enhanced with large FLASH memory into a standard PC can highly reduce the computation time of seed-based algorithms for intensive genomic comparison applications. We get, here, a speed-up of 75. Another way to compare performances is to say that the computer power of this architecture is nearly equivalent to a 64-node cluster.

However, these performances need to be relativized considering the cost of the accelerator. Today (mid 2006), the price of a 2-GB NAND-FLASH component is around \$40. A Virtex-2 Pro FPGA equipped with 64 GB of FLASH memory can easily fit into a single PCI board. The price of such a board would not exceed the price of the host station. In other word, with the same amount of money, the FPGA/FLASH alternative provide a substantial speed-up for this class of algorithms.

In order to compare the computational power of this FPGA architecture with other FPGA solutions we introduce a measurement unit: the number of Kilo Amino Acids (Kaa) compared to the number of Mega nucleotids (Mnt) performed every second. On the above application, we get:

$$\frac{132000Kaa \times 3200Mnt}{936 \times 10^3 sec} = 451KaaMnt/sec$$

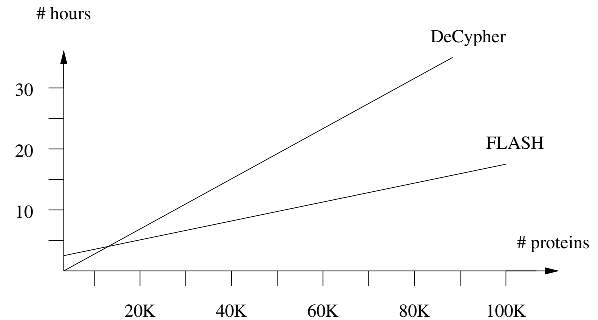

Fig. 5. Performance comparison of the DeCypher Engine and the FPGA/FLASH architecture. A database, composed of 192 bacterial genomes, is compared against set of proteins. For small sets, the DeCypher Engine is better as it is not penalized by the FLASH memory initialization. For large set, the FLASH/FPGA architecture is approximatively 2.5 times faster.

A comparison can be done with the Tera-T-BLASTN implementation on the FPGA Timelogic DeCypher Engine. The accelerator has been benchmarked for searching 4289 proteins from the E. Coli bacteria against 192 bacterial genomes (775×10^6 symbols) [1]. It took 1 hours and 36 minutes to perform the computation. From this benchmark, we can calculate the architecture performance:

$$\frac{1358Kaa \times 775Mnt}{5760sec} = 182KaaMnt/sec$$

It appears that the FPGA/FLASH architecture has a higher raw performance ratio. Actually, this unit doesn't exactly reflect the features of both architectures. The FPGA/FLASH architecture is handicaped by the initialization of the FLASH memory. Figure 5 show the comparison of an increasing set of proteins against the 192 bacterial genomes. It can be seen that

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

for small set of query proteins, the DeCyper Engine is better, even if this performance ratio is lower than the FPGA/FLASH one. This is mainly due to the time for downloading, at the begining of the computation, the index of the 192 bacterial genomes. But, as the set of the query proteins becomes larger, this penalty tends to disappear.

VI. Conclusion

In this paper, we have presented a new way of accelrating seed-based search algorithms in the context of intensive genomic sequence comparison. Our approach relies on:

- an index database structure built from seed-based heuristics;
- a large FLASH memory tightly connected to FPGA resources;
- a parallel architecture implementing an array of 160 simple ungapped operators.

We have designed a PCI based system including 64 GBytes of FLASH memory connected to a Xilinx Virtex-2 Pro component. A speed-up of 75 has been measured on a real intensive genomic sequence comparison application.

Compared to other research works, which mainly process the database as a nucleic data stream, data are structured in such a way that they can be highly re-used when accessed. Performances essentially come from that point: today FPGA components house a huge potential computational power which can really be exploited if they can be fed at a consequent data rate. Magnetic storage cannot sustain such features, and hundred of GBytes of RAM or DRAM memories are no so easy to implement. FLASH memory appears as a good compromise.

The limited write cycles of the FLASH memory might be seen as a strong constraint, leading to restrictive use of the accelerator. Actually, in the context of large computation, data remains long enougth in the FLASH memory to avoid a fast wear. The FLASH memories we used (Samsung K9W8G08U1M [19]) can support 100,000 writes before any degradation in a typical application. If we reload the complete memory every hour, then we have a potential use of more than 10 years ! Furthermore, the life of the memory can be augmented by using wear-leveling mechanisms [4] which avoids writing to always the same memory blocks.

Further researche will focus on investigating implementation of other programs from the BLAST family, especially the BLASTN program in the context of detecting all the repeat sequences inside full genomes. These applications require comparing entire genomes against themselves. They take days of computation, even on large clusters. The same index technique, based on DNA seeds, can be used to structure the database, and feed a specific DNA ungapped processor array.

More sophisticated data structures are also envisioned, such as suffix trees which are intensively used in the bioinformatics area to locate identical motifs along genomes. Today, even for medium size genomes, such structures cannot fit into the main memory of computers. Having these structures stored in FLASH memories together with their appropriate query hardware mechanisms will surely contribute to eliminating the current bottleneck in the use of suffix trees for processing large genomes.

References

[1] K. Muriki, K.D. Underwood, R. Sass, RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation, In proc. IPDPS 2005: Fourth IEEE International Workshop on High Performance Computational Biology, Denver, CO, April 4, 2005.

[2] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, D.L. Wheeler, GenBank, Nucleic Acids Res., Jan 1;33(Database issue):D34-8, 2005.

[3] J. Lancaster, J. Buhler, R. Chamberlain, Acceleration of Ungapped Extension in Mercury BLAST, 7th workshop on media and streaming processors, Barcelona, Spain, November 12, 2005

[4] E. Gal, S. Toledo, Algorithms and data structures for flash memories, ACM Computing Surveys (CSUR), Volume 37 , Issue 2, 138-163, 2005

[5] P. Krishnanurthy, J. Buhler, R.D. Chamberlain, M.A. Franklin, K. Gyang, J. Lancaster, Biosequence Similarity search on the Mercury system, in Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 365-375, 2004

[6] L. Grate, M. Diekhans, D. Dahle, R. Hughey, Sequence Analysis With the Kestrel SIMD Parallel Processor Pacific Symposium on Biocomputing, Hawai, 2001

[7] D. Lavenier, D. Guytant, S. Derrien, S. Rubini, A reconfigurable parallel disk system for filtering genomic banks, ERSA'03, Engineering of Reconfigurable Systems and Algorithms , Las Vegas, Nevada, USA, 2003

[8] S. Guytant, D. Lavenier, Evaluation of anchoring scheme for fast DNA Sequence Alignment, ECCB'2003 European Conference on Computational Biology, Paris, France, 2003

[9] S.F. Altschul et al., Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acids Res., 25:3389-3402, 1997

[10] P. Guerdoux, D. Lavenier, SAMBA: Hardware Accelerator for Biological Sequence Comparison, CABIOS, vol 13, no 6, 1997

[11] D.T. Hoang, Searching genetic databases on SPLASH2, FCCM'93, IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, 1993

[12] E. Chow, T. Hunkapiller, J. Peterson, Biological Information Signal Processor, ASAP'91, International Conference on Application Specific Array Processors, Barcelona, Spain, 1991

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

[14] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Natl. Acad. Sci., 85:3244-3248, 1988

[15] O. Gotoh, An Improved Algorithm for Matching Biological Sequence, J. Mol. Biol., 162:705-708, 1982

[16] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, J. Mol. Biol. 147-195-197, 1981

[17] R.A. Wagner and M.J. Fischer, The String-to-String Correction Problem, Journal of the ACM, 21(1):168–173, 1974

[18] TimeLogic Web Site: http://www.timelogic.com

[19] K9W8G08U1M Samsung, 1G x 8 Bit NAND Flash Memory Datasheet, http://www.samsung.com