INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Hardware Integration of a Reconfigurable Accelerator for Genomic Sequence Comparison

Alexandre Cornu — Dominique LAVENIER

Livrable 1.6 - ANR PARA

June 2008

Thème BIO

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Hardware Integration of a Reconfigurable Accelerator for Genomic Sequence Comparison

Alexandre Cornu , Dominique LAVENIER

Thème BIO — Systèmes biologiques
Équipes-Projets Symbiose

Livrable 1.6 - ANR PARA — June 2008 — 16 pages

Abstract: This document describes a parallel genomic sequence comparison operator for reconfigurable platforms. It has been implemented for speeding up the time consuming parts of the index-TBLASTN program designed for comparing full genomes with large protein banks. Speed-up from 25 to 60 has been measured on real genomic data.

Key-words: Parallelisation, similarity search, indexing, TBLASTN

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Intégration matérielle d'un accélrateur reconfigurable pour la comparaison de séquences Génomiques

Résumé : Ce document décrit un operateur parallèle de comparaison de séquences génomiques pour des plates formes reconfigurables. Il a été mis en oeuvre pour accélérer les parties critiques du programme index-TBLASTN conçu pour conparer des génomes entiers avec de grandes banques de protéines. Des accélérations de 25 à 60 ont été mesurées sur des données génomiques réelles.

Mots-clés : Parallélisation, recherche de similarité, indexation, TBLASTN

Contents

1 Introduction

With the increasing amount of complete sequenced genomes [8], the need for rapidely mining these genomic data is becoming critical. One of the first task is often to compared newly sequenced genomes with protein banks in order to discover where genes are located and perform a first annotation.

This task is generally done with sequence comparison tools such as the TBLASTN program of the BLAST family [6] [5]. Even if the BLAST programs are fast, analazing large genomes can be very time consumming.

The index-TBLASTN program is a first attempt to parallelize BLAST-like programs on parallel platforms. Contrary to TBLASTN (from the BLAST family), this version doesn't aim to search large database. Instead, it focusses on comparing consequent amount of data, that is a complete genone and a protein bank [1].

Like TBLASTN, index-TBLASTN used the word heuristic to limit the search space and proceeds into 3 steps:

1. find anchor points;
2. extend with ungapped alignment;
3. refine using dynamic programming techniques.

The main difference is that instead of indexing the query (sequences from the protein bank), index-TBLASTN indexes the complet genome following the 6 reading frame and performs a dynamic indexing of the protein bank. Furthermore, the index is enhanced with neighborhood information to avoid costly random memory accesses.

The central idea is to merge the two first steps of TBLASTN into a single one for generating very rapidely significant ungapped alignments or, at least, positions between the genome and the sequences from the protein bank where a high probability of similarity occurs. In that scheme, the first part of the algorithm only manipulates indexes which represent only local view of the genomes. This is enougth to determine if it is worth to run (or not) a costly dynamic programming procedure. The great advantage is that these local computations can be highly parallized, as it is shown in the next section.

This technique has been sucessfully tested through various implementations: SSE instructions, multithreading and GPU devices [2]. This document presents an FPGA implementation through a Parallel Sequence Comparison operator (PSC operator).

Briefly, this operator is located on a reconfigurable board connected througth a PCIe interface. It is made of thousands of small dedicated processors working in parallel for computing a simple score between two short protein sequences. It output only the short sequences (or more specifically positions of the short sequences) having a relatively good similarity.

The rest of the document is organized as follows: the next section presents the index-TBALSTN algorithm. Section 3 details the Parallel Sequence Comparison (PSC) operator. The last section gives performance results.

2 index-TBLASTN Algorithm

2.1 Index

The index-TBLASTN algorithm requires first to index a genome (DNA sequence) into its 6 protein reading frames. This is done offline. Results are stored into two specific files:

- genome.dat: this file contains a sorted list of seeds with their neighborhood;
- genome.idx: this file contains a list of 20^W integers corresponding to the seed number of elements of size W.

As an example, suppose the following DNA sequence S:

```
S = A T G G A C C A G G A T A G G A C C A C G A G T A G
    |                   |                   |
    0                   10                  20
```

Translation into the 3 reading frames gives:

```
M D A D R T T S  (frame 1)
W T R I G P R V  (frame 2)
G P G - D H E -  (frame 3)
```

The reverse strand of S is :

```
C T A C T C G T G G T C C T A T C C T G G T C C A T
          |                   |                   |
          20                  10                  0
```

Translation into the 3 reading frames gives:

```
L L V V V S W S  (frame -1)
Y S W S Y P G P  (frame -2)
T R G P I L V H  (frame -3)
```

With a size seed equal to 1 amino acid ($W = 1$) and a neighborhood of 3 amino acids, we get the intermediate index:

```
    frame 1             frame 2             frame 3

*** M DAD   0       *** W TRI   1       *** G PG-   2
**M D ADR   3       **W T RIG   4       **G P G-D   5
*MD A DRT   6       *WT R IGP   7       *GP G -DH   8
MDA D RTT   9       WTR I GPR  10       GPG - DHE  11
DAD R TTS  12       TRI G PRV  13       PG- D HE-  14
ADR T TS*  15       RIG P RV*  16       G-D H E-*  17
DRT T S**  18       IGP R V**  19       -DH E -**  20
RTT S ***  21       GPR V ***  22       DHE - ***  23
```

```
   frame -1           frame -2           frame -3

*** L LVV  -25      *** Y SWS  -24      *** T RGP  -23
**L L VVV  -22      **Y S WSY  -21      **T R GPI  -20
*LL V VVS  -19      *YS W SYP  -18      *TR G PIL  -17
LLV V VSW  -16      YSW S YPG  -15      TRG P ILV  -14
LVV V SWS  -13      SWS Y PGP  -12      RGP I LVH  -11
VVV S WS*  -10      WSY P GP*   -9      GPI L VH*   -8
VVS W S**   -7      SYP G P**   -6      PIL V H**   -5
VSW S ***   -4      YPG P ***   -3      ILV H ***   -2
```

The final index is sorted following the seed alphahbetical order:

```
 genome.idx      genome.dat

 A 1              0: *MD A DRT    6      24: DAD R TTS   12
 C 0              1: **M D ADR    3      25: IGP R V**   19
 D 3              2: MDA D RTT    9      26: **T R GPI  -20
 E 1              3: PG- D HE-   14      27: **Y S WSY  -21
 F 0              4: -DH E -**   20      28: RTT S ***   21
 G 5              5: *** G PG-    2      29: VSW S ***   -4
 H 2              6: *GP G -DH    8      30: VVV S WS*  -10
 I 2              7: *TR G PIL  -17      31: YSW S YPG  -15
 K 0              8: SYP G P**   -6      32: ADR T TS*   15
 L 3              9: TRI G PRV   13      33: DRT T S**   18
 M 1             10: G-D H E-*   17      34: **W T RIG    4
 N 0             11: ILV H ***   -2      35: *** T RGP  -23
 P 5             12: WTR I GPR   10      36: GPR V ***   22
 Q 0             13: RGP I LVH  -11      37: LLV V VSW  -16
 R 4             14: *** L LVV  -25      38: *LL V VVS  -19
 S 5             15: GPI L VH*   -8      39: PIL V H**   -5
 T 4             16: **L L VVV  -22      40: LVV V SWS  -13
 V 5             17: *** M DAD    0      41: *** W TRI    1
 W 3             18: TRG P ILV  -14      42: VVS W S**   -7
 Y 2             19: WSY P GP*   -9      43: *YS W SYP  -18
                 20 :YPG P ***   -3      44: *** Y SWS  -24
                 21: **G P G-D    5      45: SWS Y PGP  -12
                 22: RIG P RV*   16      46: GPG - DHE   11
                 23: *WT R IGP    7      47: DHE - ***   23
```

The genome.idx file contains 20^W entries, corresponding to the 20^W different possible seeds of size W. For each seed, it indicates the number of seeds in the genome.dat file.

2.2 General Algorithm

When comparing a genome (already indexed) to a protein bank, the index-TBLASTN algorithm can be split into 2 steps. The algorithm can be described as follows:

```
 1:  for i=0 to MAX_SEED reset(INDEX_PROT[i])             // step 1
 2:  for all sequences sq in the protein bank
 3:    for all seeds sd of ss
 4:      add_index(INDEX_PROT[sd])
 5:      if size(INDEX_PROT[sd]) = MAX_ELMT
 6:        get_index(INDEX_GENOME,sd)
 7:        ungapped_alignments(RESULTS,INDEX_GENOME,INDEX_PROT[sd])
 8:        add_align(UNGAPPED,RESULTS)
 9:        reset(INDEX_PROT[sd])
10:
11:  for all elements x of UNGAPPED                      // step 2
12:    compute_align(x)
```

`INDEX_PROT[k]` is an index structure of limited size located into the main memory of the computer. It has the same structure as the genome index but, for a specific seed, it can contain a maximum of `MAX_ELMT` elements (one element beeing a seed, its neighborhood and its position to the genome).

The protein bank is scanned only once and an index is dynamically built (line 4). When, for a seed, the maximum value is reached (line 5), the corresponding genome seed index is get (line 6). From these two indexes, ungapped alignments are performed. Results are stored into the `UNGAPPED` structure which memorizes only alignments having exceeded a given threshold.

The second step consists in extending ungapped alignments by using more sophisticated techniques such as dynamic programming.

In this document, we focus only on step 1 which represents a large percentage of the overall execution time. More specifically, code profiling shows that the `ungapped_alignment` procedure (line 7) represents more than 98% of step 1.

2.3 Ungapped Alignment Procedure

The `ungapped_alignment` procedure takes as input 2 index structures: one from the genome and the other from the protein bank. Both indexes are related to the same seed. If N is the number of elements in the genome index and M the number of elements in the protein bank index, then the procedures computes $N \times M$ scores based on an amino acid substitution matrix. The pseudo C-code is the following:

```
void ungapped_alignment(RESULTS, INDEX_GENOME, INDEX_PROT)
 {
   for all elements G of INDEX_GENOME {
       for all elements P of INDEX_PROT {
           score = 0;
           maxi  = 0;
           for (i=0; i<SIZE_SEED+2*SIZE_NEIGHBORHOOD; i++) {
               score = score + SUB(G.seq[i],P.seq[i]);
               if (score<0) score = 0;
               if (maxi<score) maxi = score;
           }
       }
       if (maxi>=THRESHOLD) add_results(RESULTS,G.pos,P.pos);
   }
```

```
}
```

The notation `G.seq[i]` represents the i^{th} amino acid of the sequences made of the concatenation of the left neighborhood, the seed and the right neighborhood of the element G. Similarly, `G.pos` represents the position of the seed inside the genome. At the end of the procedure, the structure `RESULTS` contains a list of position pairs corresponding to the locations where significant similarities have been detected. From these *anchoring points* the second step can be launched.

2.4 Parallelism

From the code of the `ungapped_alignment` procedure it can be seen that the computations performed inside the 2 nested `for all` loops are independant. Thus, the $N \times M$ score calculations can be attached to a specific task and run independantely.

The parallel operator describes in the next section is based on this observation.

3 Parallel Sequence Comparison Operator

3.1 Functionality overview

The PSC operato) aims to hardwire onto a reconfigurable platform the computation of the `ungapped_alignment` procedure. This operator receives two data flows, the `INDEX_GENOME` and `INDEX_PROT` structures and output pairs of positions corresponding to significant ungapped alignments.

3.2 Connection to the host processor

The operator is implemented on a reconfigurable board connected throught a PCIe interface. The figure 1 details the way the input/output of the PSC operator I/O are linked.

The two input are connected to two input FIFOs which are feeded by two **read actors**. These two components are initialized by the host processor to performed DMA transfers from the host memory.

Similarly, the output of the PSC operator is connected to a FIFO which feed a **write actor**. This device is also initialized by the host processor for DMA transfers.

In that scheme, data transfers are initiated by the host. The operator starts working as soon as data enter the input FIFOs. The operator stops when the last data is read from the input FIFOs. To detect the end of the process, the FIFO are enhanced with a data flag indicating the data status. Thus, when the last data is sent to the FIFOs, the read actor set this flag.

3.3 Operator Architecture

The PSC operator architecture has been designed to be able to drive data to and from a large number of processing elements (PE). All the PEs are working

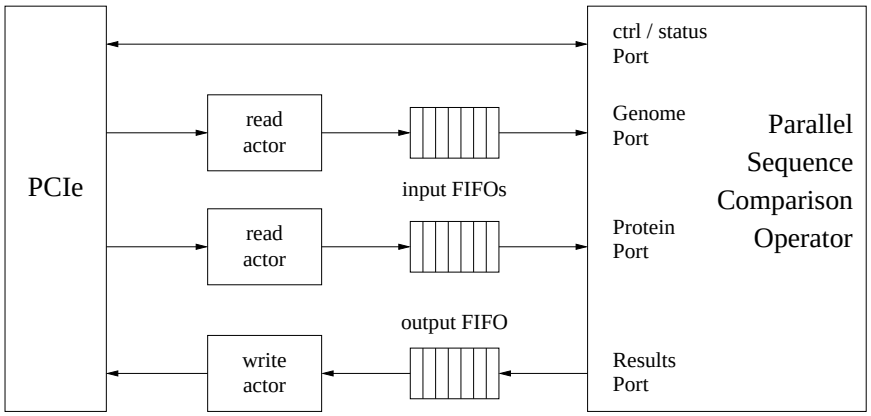

Figure 1: Connection of the PSC operator to the host processor througth PCIe interface and read/write DMA actors

in a parallel way as a Simple Instruction Multiple Data architecture. Each PE computes the distance between 2 sub-sequences of amino-acids.

The design needs to run at 250 MHz to match PCIexpress frequency. This high frequency requires to prefer short and parallel data paths, rather than long and shared paths. This has led to develop a pipeline structure.

Indeed, PEs (or clusters of several PEs) are separated by register barriers, which delay and reenforce all signals (data and control signals). This structure, that delayed the computation process from a PE to the next one, makes the use of feedback signal impossible: PEs are entirely under the control of a master unit. Figure 2 represents this architecture.

The design is independant of the number of PEs. This enable to use a single PE for first validation (architecture simulation, software development and validation,...), then gradually increase the pipeline length and finally use a maximum of PE for highest acceleration (maximum PE number depends on FPGA ressources). The PSC Operator is actually working with 128 PEs (tests are currently done to have 256 PEs but timing contraints make it difficult).

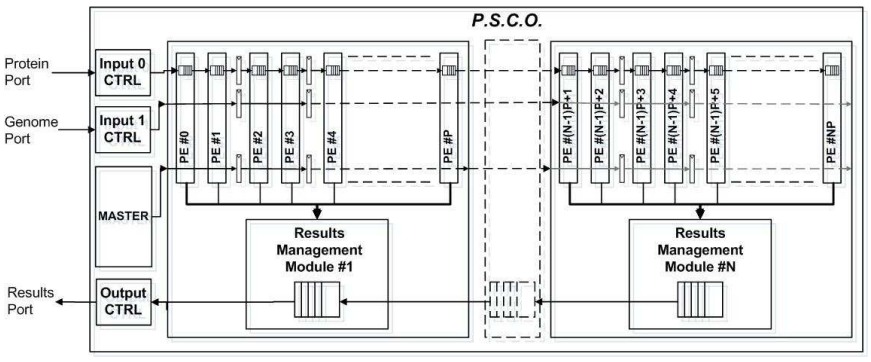

Figure 2: PSC Operator Arcitecture

PSC operator architecture is divided into the following components:

- *Input Controler 0*: reads query sub-sequences on protein port 0 and pushes them in the data pipeline 0.
- *Input Controler 1*: reads indexed genome sub-sequences on genome port and pushes them in the data pipeline 1.
- *PE Slots*: PE Slots consists of groups of PE with a common result management module. The factorization of Result management modules enable to save FPGA ressources. PE Slots are made of:
  - *PE*: Processing Element that stores a query sub-sequences in order to reuse this data as much as possible and computes distance between stored query sub-sequences and new genome sub-sequences (see Processor Description section).
  - *Result Management Module*: scan results from a cluster of PE and store them into a Fifo if the distance computation result is higher than a predifined threshold. These Fifo are cascaded so that data join the Output Controler.
- *Output Controler*: reads results from cascaded Fifo and writes them on Result port.
- *Master controler*: manages the global process (process start, data loading, computation, results recovering, end of computation).

Each Input Controler has a Sequence Counter, so that a PE, knowing its position in the pipeline, can deduce the sequence identifier from this counter value. Results returned to the host consists of 32-bit-words of two sequence identifiers and some warning flags.

3.4 Processor Description

Figure 3 represents the PE architecture. First, during the load process, it stores a query sub-sequence into a shift-register Fifo. The lookback kept that sequence stored so that it can be reused for several computations. Then, during computation process, this sequence (query) is sent AA by AA to the processor, which computes a distance against every single sequence coming from the Genome Port controler (seq bank).

During a computation (N clock cycles for a computation between 2 sequences of size N), distance between each couple of AA comes from a local RAM initialized with BLOSUM matrix values. These value are added. The maximum is stored as a result, which is then compared to the predefined treashold by the Result Management module, and finally kept or thrown away.

4 Performance evaluation

4.1 Hardware platform

The card is a XpressFX Card from PLDA, which has a 8 lanes PCIexpress interface giving a theorical bandwidth of 2.5GBytes/s.The main components of the XpressFX card are:

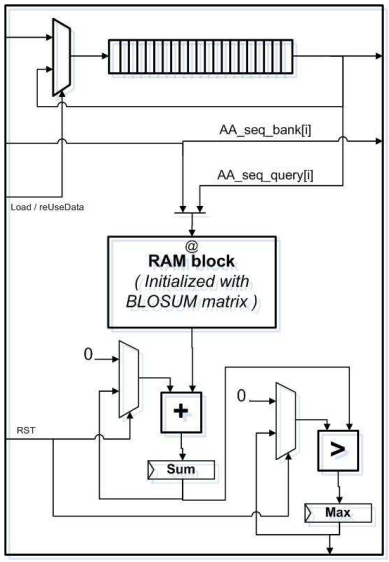

Figure 3: Processing Element Architecture

- a FPGA: VIRTEX 4 XC4FX100 FF1152 from Xilinx;
- a couple of512 MBytes DDR-SDRAM.

Figure 4: XpressFX Card from PLDA

The XpressFX card is plugged in a X86 workstation having the following features:

- Processor : Intel(R) Pentium(R) D CPU 2.80GHz
- Processor Cache : 1 Mo
- Memory : 3 Go
- Operating System : Red Hat Enterprise Linux AS release 4

4.2 Data Set

The data set has been chosen according to a real bioinformatics application dealing with mitochondrial deseases [4]. The start of the study was the comparison of the human genome with 600,000 procaryotic proteins. Here, we select a 5 subsets of these proteins as follows:

- P1K: 1000 proteins (300,005 amino acids)
- P3K: 3000 proteins (811,411 amino acids)
- P10K: 10,000 proteins (2,984,635 amino acids)
- P30K: 30,000 proteins (6,868,965 amino acids)

The chromosomes of the human genome come from the Golden Database (NCBI Build 36.1, Mar. 2006 – hg18). The following table gives, for each chromosome, its length in nucleotides.

chromosome	length	chromosome	length
1	247249719	13	114142980
2	242951149	14	106368585
3	199501827	15	100338915
4	191273063	16	88827254
5	180857866	17	78774742
6	170899992	18	76117153
7	158821424	19	63811651
8	146274826	20	62435964
9	140273252	21	46944323
10	135374737	22	49691432
11	134452384	X	154913754
12	132349534	Y	57772954

4.3 Experiments

Experiments consist in measuring the execution time with and without the PSC operator using the different data sets. In the 5 next subsections the time is reported in second and measured with the linux command `time` (there are no more jobs running on the platform).

Tests are currently performed on the shortest chromosome (21) and to a medium size chromosome (9) with a PSC operator of 128 PEs.

Chr21 and Chr9 vs P1K

chr	host	PSC op	speed-up
9	4728	190	25
21	1603	63	25

Chr21 and Chr9 vs P3K

chr	host	PSC op	speed-up
9	12266	321	38
21	4143	71	58

Chr21 and Chr9 vs P10K

chr	host	PSC op	speed-up
9	45545	1097	45
21	15144	255	59

Chr21 and Chr9 vs P30K

chr	host	PSC op	speed-up
9	103878	2652	39
21	34590	620	55

These first results are preliminary results. More experiments need to be done, especially on larger data.

Besides, two other test experiments will be performed before the end of the PARA project:

- test on a larger processor array: a 256 processor array is currently developped.
- test on a design including a memory closely connected to the PSC operator in order to reduce the PCIexpress data transfer.

References

[1] Nguyen, V. A., Lavenier, D., Speed up Subset Seed Algorithm for Intensive Protein Sequence Comparison, 6th International Conference on research, innovation & vision for the future, Ho Chi Minh Ville, Vietnam, 2008

[2] Nguyen, V. A., Lavenier, D., Fine-grained parallelization of similarity search between protein sequences, INRIA Research report, no. 6513, april 2008.

[3] Lavenier, D., Xinchun, L., Georges, G., Seed-based genomic sequence comparison using a FPGA / FLASH accelerator, International IEEE Conference on Field Programmable Technology (FPT), Bangkok, Thailand, 2006.

[4] O. Glorieux, M. Ferré, A, Fouilloux, I. Dupays, D. Raux, D. Lavenier, Y. Malthièry, P. Raynier, Y. Tourmen, , Optimisation of the NCBI-BLAST code for high throughput in silico comparative genomics in the DEISA project, JOBIM 2005, Lyon, 2005

[5] 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.

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

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

[8] Genome online database - http://www.genomesonline.org/

Annex 1: Ungapped Alignment Procedure Code

Original code

```
int ComputeDistance(ff,rq)
     FILE *ff;
     REQUEST *rq;
{
  int i,i1,i2,j,k,r,score,maxi,idx_nt;
  char s[256];
  REC_OUT DT[SIZE_BUF];

  r=0;
  for (j=0; j<rq->Qry->nb; j++)
    {
      fseeko(ff,rq->offset,SEEK_SET);
      for (i1=0; i1<rq->nbelt; i1=i1+SIZE_BUF)
        {
          i2 = SIZE_BUF;
          if (i1+i2 >= rq->nbelt) i2 = rq->nbelt - i1;
          fread (DT,i2*sizeof(REC_OUT),1,ff);
          for (i=0; i<i2; i++)
            {
              score = 0;
              maxi = 0;
              for (k=0; k<SIZE_BLOCK; k++)
                {
                  score = score + MATRIX[CODE_AA[rq->Qry->seq[j][k]]][CODE_AA[DT[i].seq[k]]];
                  if (score<0) score = 0;
                  else if (maxi<score) maxi = score;
                }
              if (maxi > X1)
                {
                  rq->Res[r].idx_nt = DT[i].idx;
                  rq->Res[r].idx_prot = rq->Qry->idx[j];
                  rq->Res[r].num_seq_prot = rq->Qry->num_seq[j];
                  r++;
                }
            }
        }
    }
  rq->nbRes=r;
  return r;
}
```

Modified code for including the PSC operator

```
int ComputeDistanceFPGA( handle, ff, rq)
     fpga_handle handle;
     FILE *ff;
     REQUEST *rq;
{
  int j,i,k,rc,r,rt;
  int resMissed =0;
  REC_OUT DT[SIZE_BUF];
  int i2,i1;
  rt=0;

  /* initializing and copying Query data into Actor 0's memory area */
  memset(addr[0], 0x1b, SIZE_FPGA*NB_QUERY);
  for (j=0; j<rq->Qry->nb; j++){
    for (k=0; k<SIZE_FPGA; k++){
      *((char *)addr[0]+j*SIZE_FPGA+k) = FPGA_AA[rq->Qry->seq[j][k]];
      }
    }

  fseeko(ff,rq->offset,SEEK_SET);

  for(i1=0 ; i1 < rq->nbelt ; i1 += SIZE_BUF) {

    i2 = SIZE_BUF;
    if (i1+i2 >= rq->nbelt){ i2 = rq->nbelt - i1;}

    fread (DT,sizeof(REC_OUT),i2,ff);

    /*Resizing and copying indexed Genome into Actor 1's memory area*/
    adjustMemAllocActor(handle,SIZE_FPGA*i2,1);
    memcpy((void *)addr[1],&DT[0],SIZE_FPGA*i2);

    /* Actors/DMA setup */
    alen[0] = SIZE_FPGA*NB_QUERY;
    alen[1] = SIZE_FPGA*i2;
    alen[2] = mlen[2];
    for (i=0 ; i<ACTS ; i++) {
      rc = fpga_act_request(handle, act_id[i], 0, alen[i], 1, 1);
      }

    /* Actors/DMA start */
    for(i=ACTS-1;i>=0;i--){
      rc = fpga_act_start(handle, act_id[i]);
      }

    /* Waiting for completion... */
    for (;;) {
      int alldone = 1;
      int sct = fpga_wait(handle, 2000);
      if (sct !=0) {
        int status;
        int rc;
```

```
        for (i=0 ; i<ACTS ; i++) {
          rc = fpga_act_get_status(handle, act_id[i], &status);
          printf("Actor %d status: %08x\n", act_id[i], status);
          if ((status & (1<<0)))  alldone = 0;
          }
        }
        if (alldone) break;
      }

    /* copying actor 2's memory data into Results structure */
    r=0;
    int idfpga0=0;
    int idfpga1=0;
    while(*((int *)addr[2]+2*r)!=0){

      idfpga0 = (*((unsigned char *)addr[2]+8*r+1))*256+(*((unsigned char *)addr[2]+8*r))-1;
      idfpga1 = (*((unsigned char *)addr[2]+8*r+3))*256+(*((unsigned char *)addr[2]+8*r+2))-1;

      rq->Res[rt].idx_nt = *((int *)((unsigned char *)addr[1]+(48*(idfpga1))+44)); //FULL_INDEX[i].idx;
      rq->Res[rt].idx_prot = rq->Qry->idx[idfpga0];                                //rq->Qry->idx[j];
      rq->Res[rt].num_seq_prot = rq->Qry->num_seq[idfpga0];                        //rq->Qry->num_seq[j]
      r++;rt++;
      }
    memset(addr[2], 0x00, 8*r);

    rc = fpga_act_hold(handle, 0);
    rc = fpga_act_hold(handle, 1);
    rc = fpga_act_hold(handle, 2);

    }

  rq->nbRes=rt;
  return r;
}
```

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399