ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

“Συ-σχεδίαση υλικού και λογισμικού της υλοποίησης NCBI-BLAST σε υβριδική πλατφόρμα υψηλής απόδοσης (HPC)”

ΡΟΥΣΟΠΟΥΛΟΣ Κ. ΧΡΗΣΤΟΣ

ΧΑΝΙΑ 2011

Contents

1. Introduction

Bioinformatics is the field of biology specialized in developing the suitable hardware and software platforms for storing and analyzing the huge amounts of data that are generated by life scientists. Sequence alignment constitutes one of the most important aspects of bioinformatics since the discovery of DNA.

1.1 Sequence alignment

Sequence alignment is a way of arranging the sequences of DNA, RNA or protein in order to identify regions of similarity. Similar sequences often derive from the same ancestral sequence, this means that if two or more sequences are similar, they probably have the same ancestor, share the same structure, and have a similar biological function. This principle works even when the sequences come from very different organisms. An accurate alignment can provide valuable information for experimentation on the newly found sequences. Sequence alignment is indispensable in basic research as well as in practical applications such as pharmaceutical development, drug discovery, disease prevention and criminal forensics.

Sequence alignment aims at identifying regions of similarity between two DNA or protein sequences (the query sequence and the subject or database sequence). Alignment's classification can be based either on completeness or on the number of sequences for alignment. In the case of the number of sequences an alignment will be distinguished in pairwise when sequences are aligned in pairs or in multiple sequence alignment (MSA) when three or more sequences are participating in the alignment. When it comes down to completeness an alignment can be global or local. Global sequence alignment targets to find the best overall alignment among the sequences, on the other hand, local's target is to find short regions of highly conserved sequence, Figure 1.1

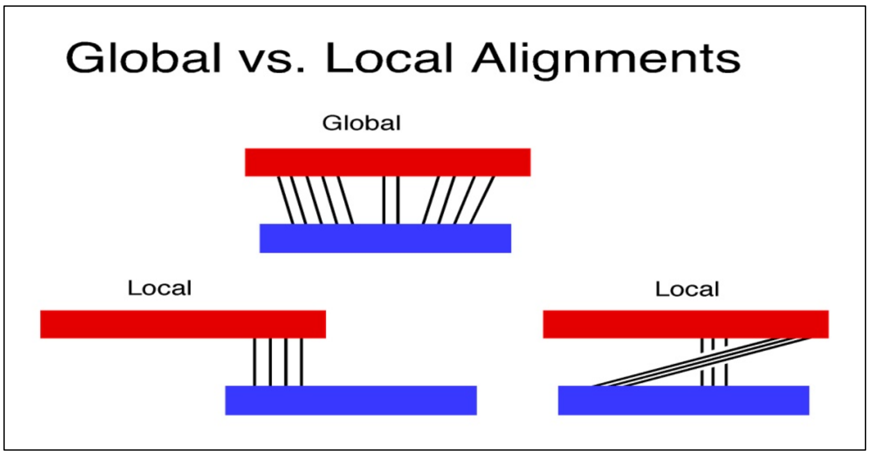


Figure 1.1: Global vs local alignment. Source http://www.pitt.edu/~mcs2/teaching/biocomp/tutorials/global.html

1.2 Sequence alignment algorithms

Since the high importance of sequence alignment has been indicated, several algorithms have been developed to offer a solution to it. These algorithms are divided in two categories, the first category is composed of the algorithms based on the dynamic programming method, providing the highest accurate answers despite the fact that their high execution time makes it’s use forbidden regarding current huge amount of data. The most known algorithms of this category are Smith-Waterman for local alignment and the Needleman-Wunch for global alignment. On the contrary, the second category is constituted of algorithms based on a heuristic approach to find the answer. Even though these answers are not as accurate as the first ones, these algorithms are more preferable due to their higher speed. BLAST and FASTA are representative algorithms of that category. When it comes to MSA a variety of methods producing both local and global alignments have been developed, which are categorized into four types, Dynamic programming, Progressive, Iterative and Motif finding methods. Some of the most known algorithms belong in this types as shown in Figure 1.2 are ClustalW, MSA, Praline, MMER, and MEME. Because MSA is out of the scope of this thesis, there will not be an extensive analysis of its methods.

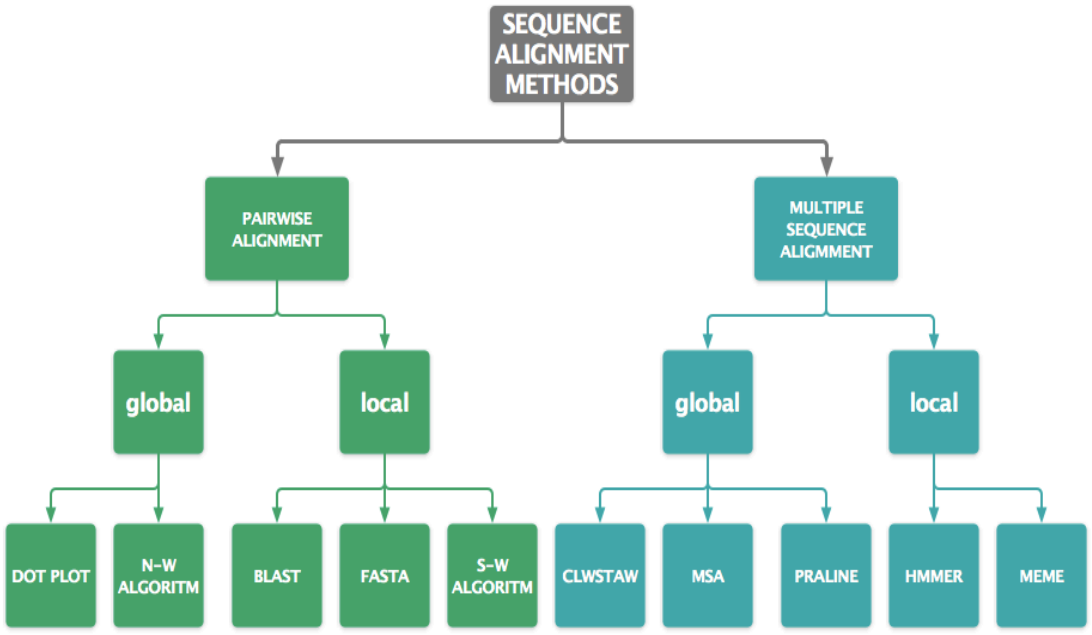
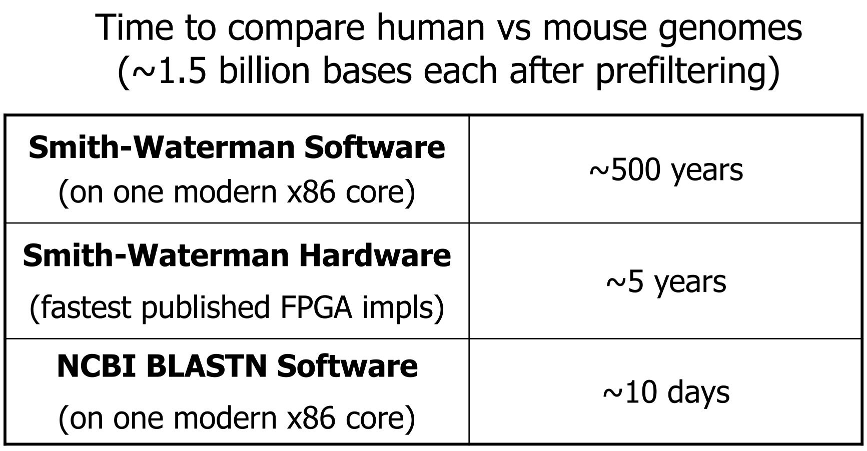


Figure 1.2: Sequence alignment methods



Time to compare human vs mouse genomes
(~1.5 billion bases each after prefiltering)

Smith-Waterman Software (on one modern x86 core)	~500 years
Smith-Waterman Hardware (fastest published FPGA impls)	~5 years
NCBI BLASTN Software (on one modern x86 core)	~10 days

Figure 1.3: Time to compare human vs mouse genoms Source Mercury BLASTN: Fast Streaming DNA Sequence Comparison ***Jeremy Buhler****, Joe Lancaster, Arpith Jacob, and Roger Chamberlain Washington University in St. Louis †BECS Technology, Inc.*

1.3 Blast algorithm

BLAST (Basic Local Alignment Search Tool) is the most powerful and most known algorithm for sequence alignment among the community of life scientists. It takes a query sequence and aligns it with every subject sequence in a database, looking for segments of high degrees of similarities. It picks out from the database those sequences that contain a segment so similar to a part or the entire query that such similarity is deemed statistically significant.

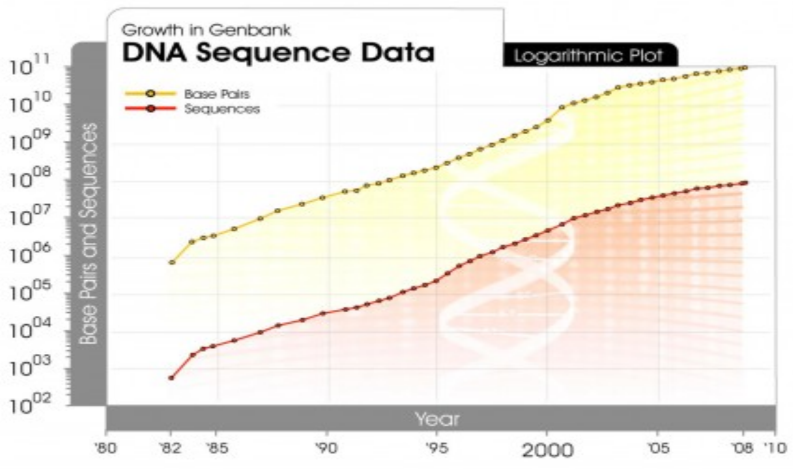
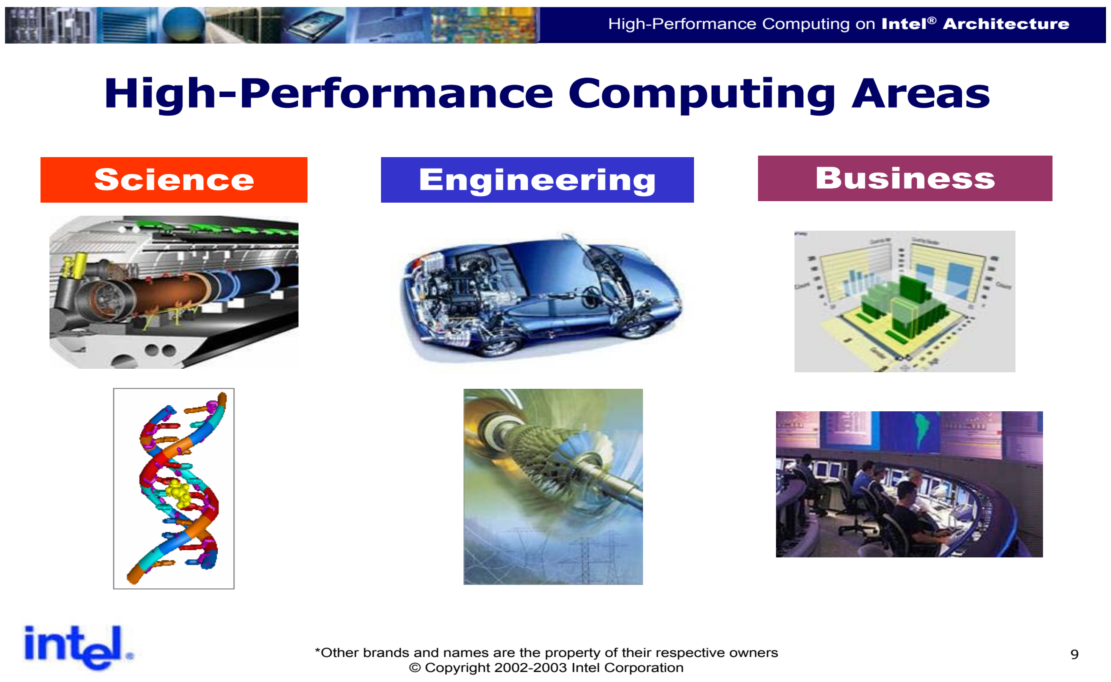


Figure 1.4: Growth in Genbank (DNA Sequencing Data). Source http://www.kurzweilai.net/dna-sequencing-data

Due to the exponential growth of the databases (From 1982 to the present, the number of bases in GenBank has doubled approximately every 18 months. As of 15 August 2011, GenBank release 185.0 has 130,671,233,801 bases [1]). A powerful computer system dedicated to running BLAST has been established at the National Center of Biotechnology Information, which has over five hundred thousands query submissions on a daily base, a number which soon will look very small as the query submissions per day, are being double approximately every year, which explains why sequence alignment and particularly BLAST becomes an excellent candidate for the high performance computing field.


High-Performance Computing on **Intel® Architecture**

High-Performance Computing Areas

Science	Engineering	Business

intel.
*Other brands and names are the property of their respective owners
© Copyright 2002-2003 Intel Corporation
9

Figure 1.5: High performance computing areas. Source: High-Performance Computing on Intel® Architecture ***11-Feb-2003 Toulouse, France***
Dr. Herbert CORNELIUS Dr. Bob KUHN Fabien ESDOURUBAIL Jamel TAYEB Gilbert CHAMPOUSSIN

1.4 High Performance Computing (HPC)

High performance computing is the branch of computer science that focuses on developing supercomputers and software to run on supercomputers. A fundamental area of this field is developing parallel processing algorithms and software, programs that can be divided into little pieces so that each piece can be executed simultaneously by separating the processing unit, which can be a processor, a general purpose unit or even a specific designed hardware.

High performance computing is growing with fast steps nowadays, the reason for this is the increase of computational performance necessary for many scientific applications, such as applied mathematics, economics, environmentals, bioinformatics and many more. The main reason behind this demand is the exponential growth of information and data. That's why high performance computing is a primary need for the future of science.

Until the early 2000s, general purpose single-core CPU-based systems were the processing systems of choice for HPC applications. In the mid-2000s, the HPC industry has gone through a historical step change to meet high-performance demands, General-purpose CPU vendors changed course to rely on multicore architectures. The technique of simply scaling a single-core processor's frequency for increased performance has run its course, because as frequency increases, power dissipation escalates to impractical levels.

The shift to multicore CPUs forces application developers to adopt a parallel programming model to exploit CPU performance. Even using the newest multicore architectures, it is unclear whether the performance growth expected by the HPC end user can be delivered, especially when running the most data and compute intensive applications [2]. Today the idea of heterogeneous computing is shaping the present and future of HPC.

Generally speaking, a heterogeneous computer is a system that uses different types of computational units to accomplish the work of the applications that use the computer. The basic idea is coupling to a multicore CPU, a new kind of computational engine, such as a field-programmable gate array (FPGA), a Cell processor, a graphics processing unit (GPU) or even a combination of these. This infrastructure becomes a heterogeneous computer with huge processing power and lower power consumption. In this configuration the added computational units accelerate several calculations of interest, and so are often referred to as accelerators.

1.5 Thesis contribution

The contribution of this thesis is the analysis of the software implementation of the Blast algorithm provided by NCBI. Although we spent time on BLAST algorithm's ideas, most of our work was on a specific version of it, the nucleotide BLAST algorithm(BLASTn). We also developed two hardware software co-design implementations based on it and mapped them in a state of the art heterogeneous computing system. These two implementations are different steps of continuous development and evolution resulting in a functional prototype of BLASTn application running on our system. We achieved our goals concerning training, experience gaining with this complicated high performance system and fidelity of our implementation.

1.6 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 briefly describes BLAST algorithm and focuses mostly on NCBI BLASTn implementation. Chapter 3 describes the HC-1 hybrid super computer system in details. Chapter 4 has the first and second generation architectures for BLASTn. Chapter 5 has performance results and source usability of the second generation implementations Finally, Chapter 6 has future work proposals and some conclusions from this work.

2. BLAST Algorithm

BLAST (Basic Local Alignment Search Tool) is the heuristic search algorithm for finding most of the highest-scoring (disjoint or at least minimally overlapping) local alignments between a query and one or more database sequences. We can simply think of BLAST being for nucleotide and protein sequences what GOOGLE is for the Internet.

This chapter is organized as follows. Section 2 analyzes the basic stages of the Blast Algorithm. Section 3 describes the substitution matrices and their usage. Finally, the rest of the chapter describes the well-known NCBI-BLAST suite and the software implementation of NCBI-Blastn algorithm as well as, its parameters.

2.1 Blast algorithm stages

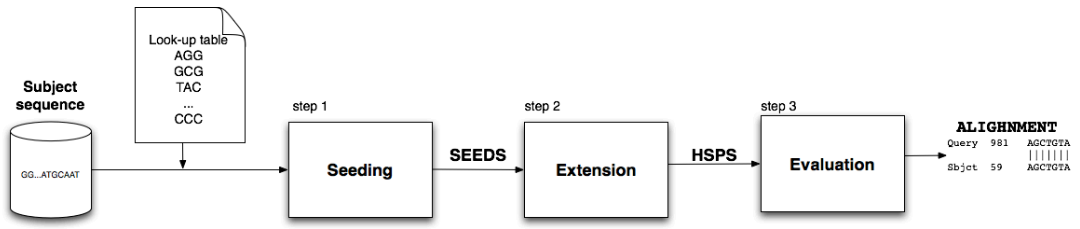


Figure 2.1: BLAST algorithm steps

BLAST search consists of three steps: seeding, extension, and evaluation.

During the first step, the query sequence is split forming a list (Look-up table), which contains all the continuous overlapped subsequences of length W, which are called w-mers. For example if sequence ACGTAAATGCAG is the query of length 12 and W is equal to 3, the formed list will contain 10 W-mers.

When the list is generated the database sequences are searched against it for finding all common words, which might possibly be part of a High

Score Pair (HSP), and which will be used as seed in the second step.

The basic idea behind seeding is the simplified assumption that any alignment of interest between query and database sequence will contain at least one region of W consecutive letters that is high-scored. Making the value of W too small increases the work, and making W too large reduces dramatically the work as it eliminates the search space, but also causes most alignments to be missed.

Going to the extension step, all the seeds that occurred by chance are discarded and only the seeds which are part of a larger common subsequence are kept, this is done by extending the alignment to the left and right of the seed to find the alignments whose scores are greater than a threshold S. In the initial version of BLAST presented in [3] this extension is called the one-hit method and consumes most of the processing time of the algorithm [4]. In order to improve this, the two-hit method was introduced in 1997 [5]. According to this method, the algorithm requires two hits rather than one to invoke an extension and hence the threshold parameter T must be lowered to retain comparable sensitivity. As a result, many more single hits are found, but only a small fraction has an associated second hit on the same diagonal that triggers an extension. During this process, the quality of the alignment is calculated by a scoring scheme, which is relying on scoring matrixes (Substitution Matrices), such as the popular BLOSUM62 and PAM, described in another part of this thesis. If the ungapped score is above a user-defined threshold, the seed can be used to produce a gapped alignment based on a Smith–Waterman type algorithm [6].

Finally, during the third step, BLAST determines which alignments of the previous step are statistically significant using the query and database sequence lengths, the substitution matrix and the sequence statistics. The accepted alignments are those whose probability of finding such an alignment by chance is lower than a user-defined value [7].

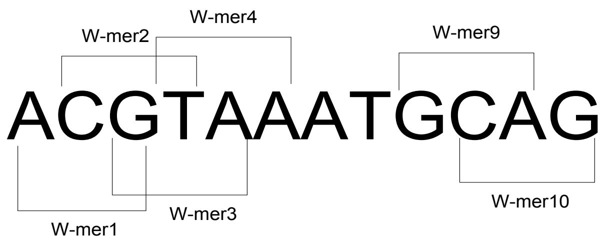


Figure 2.2 : Query w-mers of lengh 3.

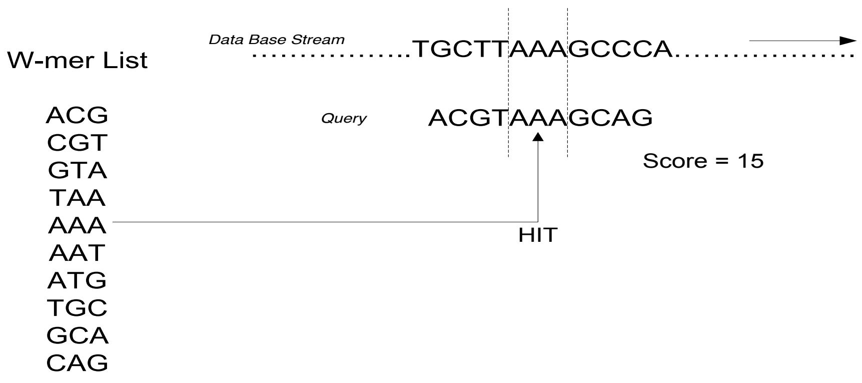


Figure 2.3: example of the Seeding step of BLAST algorithm.

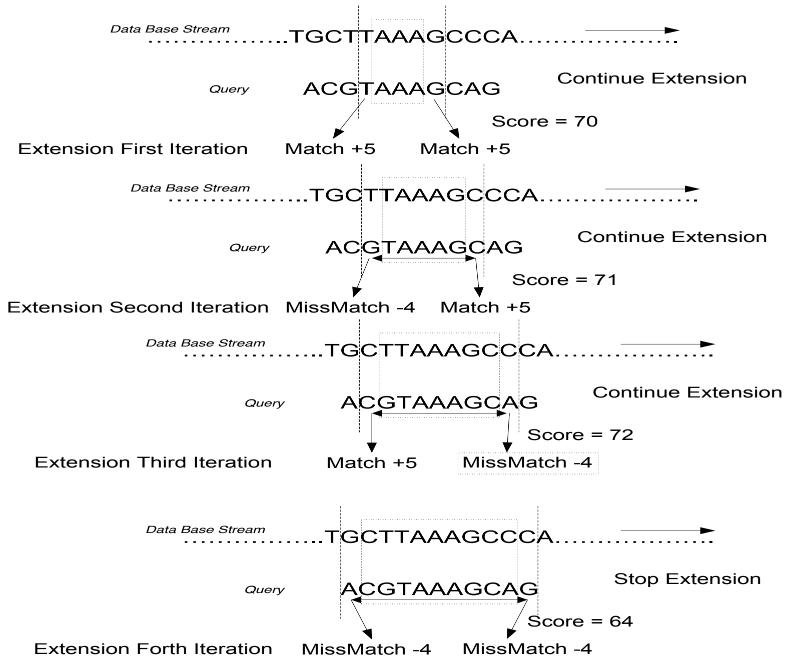


Figure 2.4: Example of extension step of BlAST algorithm.

2.1.1 Substitution Matrices

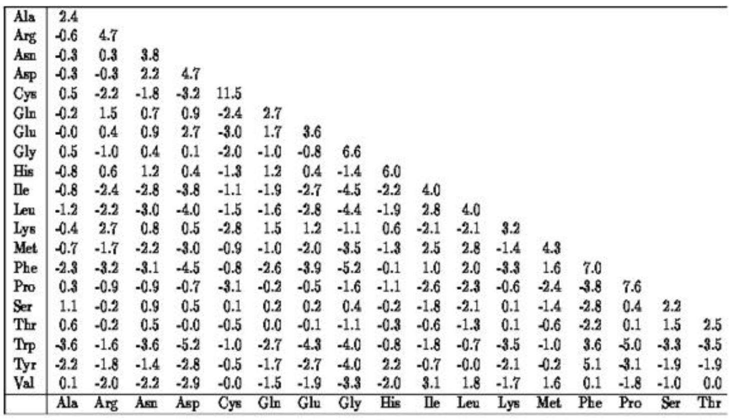
A key element in evaluating the quality of a pairwise sequence alignment, as already mentioned, is the "substitution matrix", which assigns a score for any possible aligned pair of residues. Generally, different substitution matrices are tailored to detecting similarities among sequences that are diverged by differing degrees. In case of nucleotide residues, a simple matrix, like the matrix in Table 1.1, is most of the times the proper choice because the simplicity of the matrixes of amino acids residues makes their construction difficult due to the rarity of certain amino acid substitutions. After a lot of studies, many matrices, which are proper for protein alignments, have been derived, among them the most known are PAM and BLOSUM.

PAM (Point Accepted Mutation) matrices, Figure 2.5, introduced by Margaret Dayhoff [8] in 1978 based on 1572 observed mutations in 71 families of closely related proteins. Each matrix is twenty-by-twenty (for the twenty standard amino acids); which has the score for every pair of proteins.

BLOSUM (BLOcks of Amino Acid SUbstitution Matrix), Figure 2.6, was first introduced in a paper by S.Henikoff and J.Henikoff [9], they are derived from the Blocks database, a set of ungapped alignments of sequence regions from families of related proteins. A clustering approach sorts the sequences in each block into closely related groups, and the frequencies of substitutions between these within a family, derive the probability of a meaningful substitution [10].

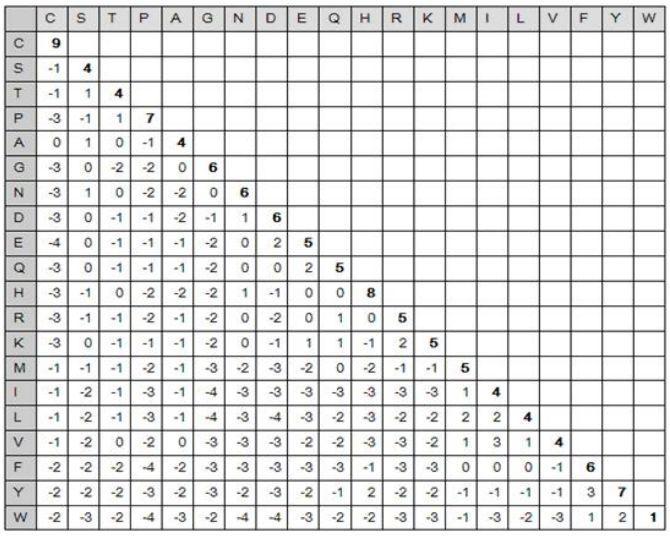
	U	**C**	**A**	**G**
U	1	-1	-1/2	-1
C	-1	1	-1	-1/2
A	-1/2	-1	1	-1
G	-1	-1/2	-1	1

Table 1 .1: Simple substitution matrix example



Ala	2.4																
Arg	-0.6	4.7															
Asn	-0.3	0.3	3.8														
Asp	-0.3	-0.3	2.2	4.7													
Cys	0.5	-2.2	-1.8	-3.2	11.5												
Gln	-0.2	1.5	0.7	0.9	-2.4	2.7											
Glu	-0.0	0.4	0.9	2.7	-3.0	1.7	3.6										
Gly	0.5	-1.0	0.4	0.1	-2.0	-1.0	-0.8	6.6									
His	-0.8	0.6	1.2	0.4	-1.3	1.2	0.4	-1.4	6.0								
Ile	-0.8	-2.4	-2.8	-3.8	-1.1	-1.9	-2.7	-4.5	-2.2	4.0							
Leu	-1.2	-2.2	-3.0	-4.0	-1.5	-1.6	-2.8	-4.4	-1.9	2.8	4.0						
Lys	-0.4	2.7	0.8	0.5	-2.8	1.5	1.2	-1.1	0.6	-2.1	-2.1	3.2					
Met	-0.7	-1.7	-2.2	-3.0	-0.9	-1.0	-2.0	-3.5	-1.3	2.5	2.8	-1.4	4.3				
Phe	-2.3	-3.2	-3.1	-4.5	-0.8	-2.6	-3.9	-5.2	-0.1	1.0	2.0	-3.3	1.6	7.0			
Pro	0.3	-0.9	-0.9	-0.7	-3.1	-0.2	-0.5	-1.6	-1.1	-2.6	-2.3	-0.6	-2.4	-3.8	7.6		
Ser	1.1	-0.2	0.9	0.5	0.1	0.2	0.2	0.4	-0.2	-1.8	-2.1	0.1	-1.4	-2.8	0.4	2.2	
Thr	0.6	-0.2	0.5	-0.0	-0.5	0.0	-0.1	-1.1	-0.3	-0.6	-1.3	0.1	-0.6	-2.2	0.1	1.5	2.5
Trp	-3.6	-1.6	-3.6	-5.2	-1.0	-2.7	-4.3	-4.0	-0.8	-1.8	-0.7	-3.5	-1.0	3.6	-5.0	-3.3	-3.5
Tyr	-2.2	-1.8	-1.4	-2.8	-0.5	-1.7	-2.7	-4.0	2.2	-0.7	-0.0	-2.1	-0.2	5.1	-3.1	-1.9	-1.9
Val	0.1	-2.0	-2.2	-2.9	-0.0	-1.5	-1.9	-3.3	-2.0	3.1	1.8	-1.7	1.6	0.1	-1.8	-1.0	0.0
	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr

Figure 2.5 : Pam 250 matrix example Source http://www.birec.org/sandbox/omamasaudtutorial



	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	**9**																			
S	-1	**4**																		
T	-1	1	**4**																	
P	-3	-1	1	**7**																
A	0	1	0	-1	**4**															
G	-3	0	-2	-2	0	**6**														
N	-3	1	0	-2	-2	0	**6**													
D	-3	0	-1	-1	-2	-1	1	**6**												
E	-4	0	-1	-1	-1	-2	0	2	**5**											
Q	-3	0	-1	-1	-1	-2	0	0	2	**5**										
H	-3	-1	0	-2	-2	-2	1	-1	0	0	**8**									
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	**5**								
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	**5**							
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	**5**						
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	**4**					
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	**4**				
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	**4**			
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	**6**		
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	**7**	
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	**1**

Figure 2.6: BLOSUM 62 matrix example. source http://www.NCBI.nlm.nih.gov/blast/html/sub_matrix.html

2.2 NCBI-BLAST

NCBI-BLAST edition is by far the most known open source implementation of the BLAST algorithm. It is actually employed by a collection of programs each one specially tailored to implement a different kind of alignments. These programs are BLASTP, BLASTN, PSIBLAST, PHIBLAST, RPSBLAST, BLASTX, TBLASTN and TBLASTX. Their differences are discussed in more details below:

- BLASTP is suitable for alignments between one or more protein *query* sequences to one or more *subject* protein sequences.
- BLASTN is the same as BLASTP but is used for nucleotide instead of protein sequences.
- *Megablast* is an optimized version of BLASTN for finding very similar alignments in very large sequences.
- PSIBLAST is used to align a protein query to a protein database, but attempts to build up a query-specific scoring model that will be sensitive only to database sequences that lie within the same 'protein family'.
- BLASTX compare the six-frame conceptual translation products of an input nucleotide query sequence (both strands) vs. a protein sequence database.
- TBLASTX searches a translated collection of nucleotide queries vs. a translated collection of nucleotide subjects. It is usually used for aligning a hypothetical protein sequences whose underlying nucleotides have diverged significantly from each other.
- RPSBLAST searches a query protein sequence or protein sequences vs. a database of position specific scoring matrices (PSSMs, profiles, or more commonly known as conserved

domains) to identify the ones the query are similar to.

- TBLASTN compares a protein sequence to the six-frame translations of a nucleotide database. It is a very productive way of finding homologous protein coding regions in unannotated nucleotide sequences.
- PHIBLAST computes local alignments between a single protein sequence and a database of protein sequences. It starts with a collection of matches, on the query and database sequences, to a specified regular expression pattern. This expression makes PHIBLAST more specific than blastp.

NCBI has established a free online service for all the available flavors of blast, and a free ftp server from where anyone can download the source code of the latest or older versions of blast suite for local machine usage, and a huge amount of preformatted and unformatted sequence databases, as well. NCBI also has developed precompiled executables of their software for almost every operating system known today.

In the current thesis, for simplicity reasons, it is the NCBI-Blastn program that is used and all the other programs are disregarded.

2.3 NCBI-BlASTn

As already mentioned, blastn program is suitable for comparing query nucleotides sequences vs. a subject nucleotide database. This algorithm is suitable for finding similar sequences, but not identicals. The process of finding similar sequences is based on generating an indexed table or dictionary of short subsequences, which are called words, for both the query and the database. One of the important parameters governing the sensitivity of BLAST searches, is the length of the initial words (word size). The most important reason that blastn is more sensitive compared to MEGABLAST, which is suitable for nucleotide sequence search also, is that it uses a shorter word size. That makes blastn more sufficient than MEGABLAST at finding alignments that are related to nucleotide sequences from other organisms since the initial exact match can be shorter. The word size is adjustable in blastn and can be reduced from the default value of 11 to a minimum of 7 to increase sensitivity. This word size can also be increased to make the search speed faster and limit the number of database hits.

2.3.1 NCBI-BLASTn usage

In this section we will provide the necessary information needed for performing a BLASTn ungapped alignment on our local machine using self compiled NCBI's blast package source code version 2.2.24 [Aug-08-2010].

Most programs in the blast package, are command line program with no graphic user interface (GUI). We control the programs through command line options issued in a terminal window. These options instruct each program what program function, query, and database to use. They also control the search sensitivity, the result format, and the name of the output file, etc. Supposing that our query file is the test.fasta containing the properly formatted query sequences (proper formats are analyzed later in this chapter) and again the proper formatted database file is the ecoli, the command performing a basic alignment is the following:

```
blastall -p blastn -d ecoli.nt -i test.fast -o output_test.out
```

Blastall is used to perform one of the five flavors of blast (blastp, blastn, blastx, tblastn and tblastx). Below the most useful command arguments of the program will be explained.

- -p flag denotes the choice of the program name. It must be followed by one of the strings "blastp", "blastn", "blastx", "tblastn", or "tblastx".
- -d flag is followed by the name of a formatted database. Multiple database names (bracketed by quotations) are also accepted. An example would be *-d "ecoli.nt est"* , which will search both the ecoli and est databases, presenting the results as if they are one database consisting of the concatenation of both.
- -i flag is followed by the input file. This flag is optional as blastall uses the standard input stream for input sequences by default. If multiple sequence entries are in the input file, all queries will be searched.
- -o flag denotes the file in which the output will be stored. The default output stream is the standard output.
- -F flag is used to specify one or more filters to be used to mask query sequence(s), it can be either true(T) or false(F) .
- -g flag guides blastall to perform gapped or ungapped alignment, it also have true(T) or false(F) input.

These are the most basic and useful arguments. The first four are used in almost every alignment, while in combination with the last two; blastall will provide an unfiltered ungapped alignment. In addition, the blastall method offers many more arguments for more flexible and more necessary specific alignments. Due to the fact that most of the alignments used by this thesis were ungapped and unfiltered we stick command below:

```
blastall -p blastn -d ecoli.nt -i test.fast -o output_test.out  -F F –g F.
```

2.3.2 Data input formats

Query input sequences, that are passed in blastall and especially blastn program with -i argument, are nucleotide sequences in FASTA format. Sequences in FASTA format consists of one line of comments beginning with a '>' symbol, followed by any number of lines, of any length of sequence information. Lines except the last one often are limited to sixty characters. Nucleotide sequences are represented in the nucleic acid codes, with these exceptions: lower-case letters are accepted and are mapped into upper-case; N may be used for an unknown nucleic acid residue. An example of a nucleotide sequence in this format is presented in figure 2.7 below:

```
>18BI1 Human MLC1emb gene for embryonic myosin
alkaline light chain, promoter and exon 1
GTGAAGAGAGAGCTGTGGCATGAAGGGGAGGGGGCTGGTGGCCCCAAACCTGG
TGACAA
TACACAGTTGTCAGCTGTACCCTGCTGGCGTTTCTTCCTTTTATAGTCAGCAG
CAGTTG
CTCTTGCTTTCACCCAGCCCCTCTGTGGGGCTCCTGCCCAGGATAAAAGGGAA
GGGAGG
CAGCCCAGGCTCCTATCTCATCTCCCAGACGCCACGTCTCTCGGTTTCTTCTT
AG
```

Figure 2.7: FASTA format example. Source http://bayesweb.wadsworth.org/gibbs/fasta.html

On the other hand, input databases, which are entered by –d argument, are not in the readable FASTA format, but in a compressed format suitable for fast execution of blastn and more efficient memory usage. Such databases are generated by NCBI's provided formatdb program. Although formatdb's usage is out of this thesis scope, we will briefly explain the most useful arguments of this program by an example.

Firstly, we are constructing a multi-fasta file containing one or more sequences we would like to include in the database, lets name it 'db_sequences'. This file is then used to construct the index for the BLAST database, lets name the database 'db_test'. Then we execute the formatdb program with the appropriate arguments. For our example the command for constructing db_test database from our db_sequence input file, will look like this:

formatdb -i db_sequences -p F -o T -n db_test

The parameters, which are used, are explained below:

- -i argument denotes that one or more filename(s) of database data follow.
- -p argument is an optional flag which is followed by T if the type of the file is protein and F if the file consists of nucleotide sequences. The default value is set to T.
- -o argument is followed by T, parses seqID and creates indexes. When it is followed by F, it does not create any indexes.

- -n argument allows a user to create BLAST databases with a different name other than the original FASTA files. This can be used in situations where the original FASTA file is not required other than by formatdb. This can help in a situation where disk-space is tight.

Under the scope of this thesis many various sized query sequences and databases have been used for tests, which have been done for profiling and output correlation reasons. These datasets were either downloaded by NCBI's web site or were random flexible sequences generated by us.

2.3.3 BLASTn output

BLAST output report can be delivered in a variety of formats. These formats include plain text, HTML and XML formatting. As most of the times the report in plain text format consists of a large number of lines a small summary follows.

The BLAST output report starts with some header information that lists the type of program used (here blastn), the version (2.0.11), and a release date (this is only an example and not the version we are using.). Also, there are references to the BLAST program, the query definition line, and summary of the database that is used, figure 2.8.

```
BLASTN 2.0.11 [Jan-20-2000]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Query= U51677
         (2575 letters)

Database: embl.fas
           442,729 sequences; 675,252,082 total letters

Searching.................................................done
```

Figure 2.8: Fist section of BLAST output.

The second section, Figure 2.9, describes the database matches found. These include a database sequence identifier, the corresponding definition line, as well as the score (in bits) and the statistical significance ('E value') for this match.

```
                                                                 Score     E
Sequences producing significant alignments:                      (bits)  Value
```

```
U51677 Human non-histone chromatin protein HMG1 (HMG1) gene, co...  4129  0.0
L38477 Mus musculus (clone Clebp-1) high mobility group 1 prote...   353  7e-95
X80457 M.musculus HMG1 gene                                          353  7e-95
U00431 Mus musculus HMG-1 mRNA, complete cds.                        353  7e-95
L08048 Human non-histone chromosomal protein (HMG-1) retropseud...   349  1e-93
                                      .
                                      .
```

Figure 2.9: Second section of BLAST output.

In the third section Figure 2.10, each alignment is preceded by the sequence identifier, the full definition line and the length of the database sequence. Next, the score is presented (in bits and the raw score) as well as the statistical significance of the match, followed by the number of identities and positive matches according to the scoring system and, if applicable, the number of gaps in the alignment. Finally the actual alignment is shown, with the query on top and the database match (Sbjct).

```
>U51677 Human non-histone chromatin protein HMG1 (HMG1) gene, complete
            cds.
            Length = 2575

 Score = 4129 bits (2083), Expect = 0.0
 Identities = 2167/2209 (98%)
 Strand = Plus / Plus


Query: 1    atgggcaaaggagatcctaagaagccgagaggcaaaatgtcatcatatgcatttttgtg 60
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1    atgggcaaaggagatcctaagaagccgagaggcaaaatgtcatcatatgcatttttgtg 60


Query: 61   caaacttgtcgggaggagcataagaagaagcacccagatgcttcagtcaacttctcagag 120
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 61   caaacttgtcgggaggagcataagaagaagcacccagatgcttcagtcaacttctcagag 120


                                      .
                                      .
```

Figure 2.10: Second section of BLAST output.

The last section, Figure 2.11, lists some information about the database which was used as well as statistical and search parameters used:

.

```
Database: embl.fas
    Posted date:  May 15, 1998  5:37 PM
  Number of letters in database: 675,252,082
  Number of sequences in database:  442,729

Lambda     K      H
    1.37    0.711     1.31

Gapped
Lambda     K      H
    1.37    0.711     1.31


Matrix: blastn matrix:1 -3
Gap Penalties: Existence: 5, Extension: 2
Number of Hits to DB: 1123218
Number of Sequences: 442729
Number of extensions: 1123218
Number of successful extensions: 86816
Number of sequences better than 10.0: 106
length of query: 2575
length of database: 675,252,082
effective HSP length: 21
effective length of query: 2554
effective length of database: 665,954,773
effective search space: 1700848490242
effective search space used: 1700848490242
T: 0
A: 0
X1: 6 (11.9 bits)
X2: 10 (19.8 bits)
S1: 12 (24.3 bits)
S2: 19 (38.2 bits)
```

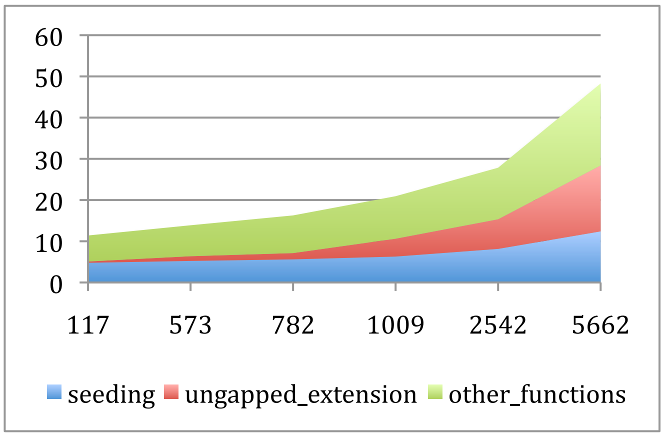
Figure 2.11: Summary of the four main parts of an NCBI_BLAST output in plain text format.

2.4 Profiling

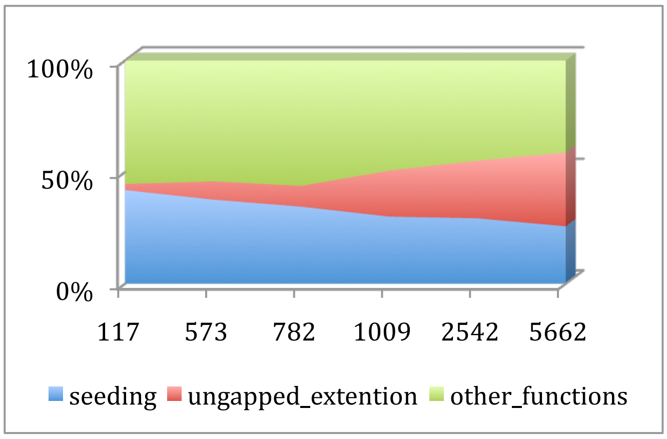
A profiling study of NCBI-BLASTN with default parameters using the open source GNU gprof profiler, is shown in the charts below. The time spent in each step of the algorithm can vary substantially with different queries and different databases. During our profiling tests we have used a variety of queries and two of the largest databases provided by NCBI :

- *Nucleotide collection (nt) which* Contains all GenBank and PDB sequences except Expressed Sequence Tags, Sequence Tagged Sites, Genomic Survey Sequences and unfinished High Throughput Genomic Sequences (all of which can be searched separately) and

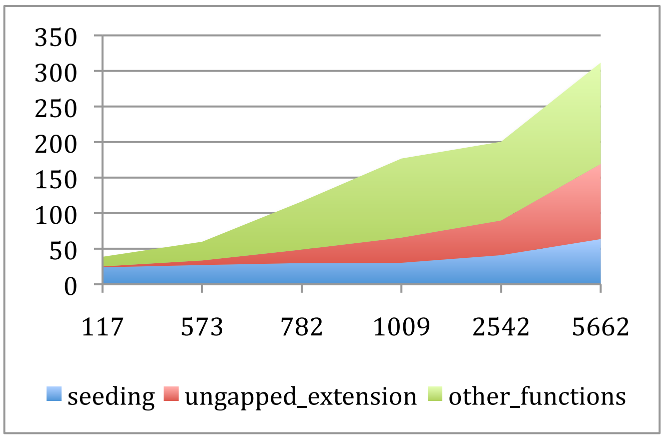
- *Environmental samples (*env_nt) Sequences from environmental samples, such as uncultured bacterial samples isolated from soil or marine samples, e.g. the Sargasso Sea project. These sequences are not in nt.



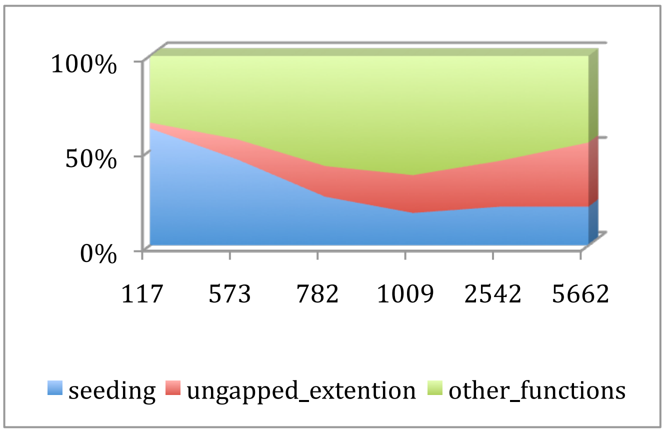
A I



A II



B I



B II

Figure 2.12: Profiling of the NCBI-BLAST code for queries of length 117 to 5662 for ungapped alignments. (**I**) charts represents real execution time. (**II**) charts represents percentage of execution time. (**A**) For env_nt database, the seeding and extension steps, respectively, consume up to 50% of the total time (blue and red). (**B**) For nt database on average, th seeding, ungapped extension, respectively, consume 55% of the total time (blue and red).

As charts in figure 2.12 reveal, the ungapped extension and mostly the seeding are the most computationally intensive parts of the NCBI-Blastn algorithm. For ungapped alignments, these two steps consume over 50% of the total execution time. Based on these observations, we decided to focus our efforts on the seeding step.

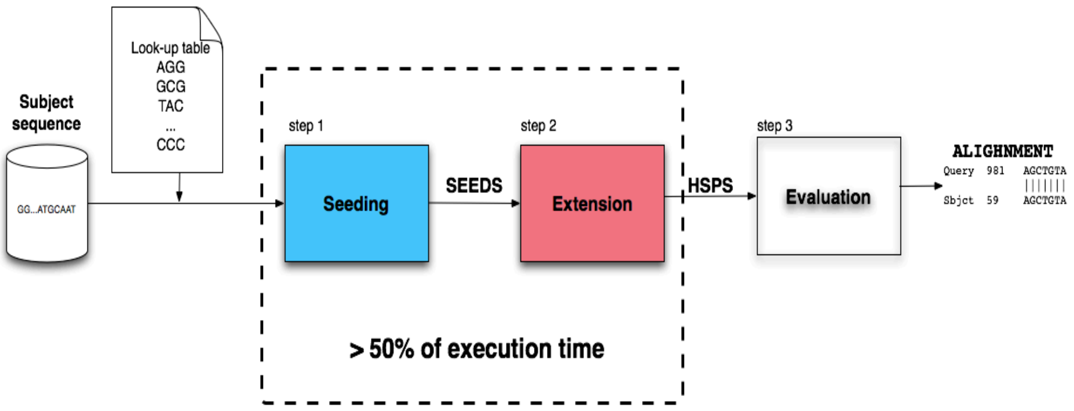


Figure 2.13 : BLAST steps and where tihe most of the execution time is spend.

2.5 Seeding step

When NCBI-BLAST finishes the initial stage for every BLAST program, which creates the Lookup table, it afterwards starts the seed collection. For this stage there are several routines named ScanSubject routines, which will touch every letter of the database being searched for most database searches. All of the routines' outputs are a full array of type BlastOffsetPair, which is a structure containing two indexes for query and database offsets of hits found. Each BlastOffsetPair represents one high-scoring alignment between a query sequence and a single subject sequence.

Generally, a ScanSubject routine can be called several times on the same subject sequence, in case there are many hits to that sequence, and it works from the beginning to the end of the sequence. There is a limit to the number of hits that a single ScanSubject routine call can retrieve, which depends on blast-program and blast look-up table. This limit is properly chosen so that when the look-up table is accessed, there will be available room for all of the query offsets at that lookup table entry. During the construction of the lookup table, an upper bound on the number of lookup entries is computed, which is given by the 'longest_chain' field of Blast lookup table structures. The maximum number of hits returned by a ScanSubject call is longest_chain plus a fixed amount.

Blast algorithm has a pool of about two dozens ScanSubject routines and every time chooses the most suitable for usage, in order to achieve maximum ScanSubject performance for nucleotide searches. The look-up table type, the word size, and the scanning stride define that choice. A word size is the number of letters, which constitute to a group of letters based on which, comparisons are made. The stride length is the number of letters between the words where the comparison begins. Figure 2.14 shows how eight letter words with 4 letter stride lengths are formed.

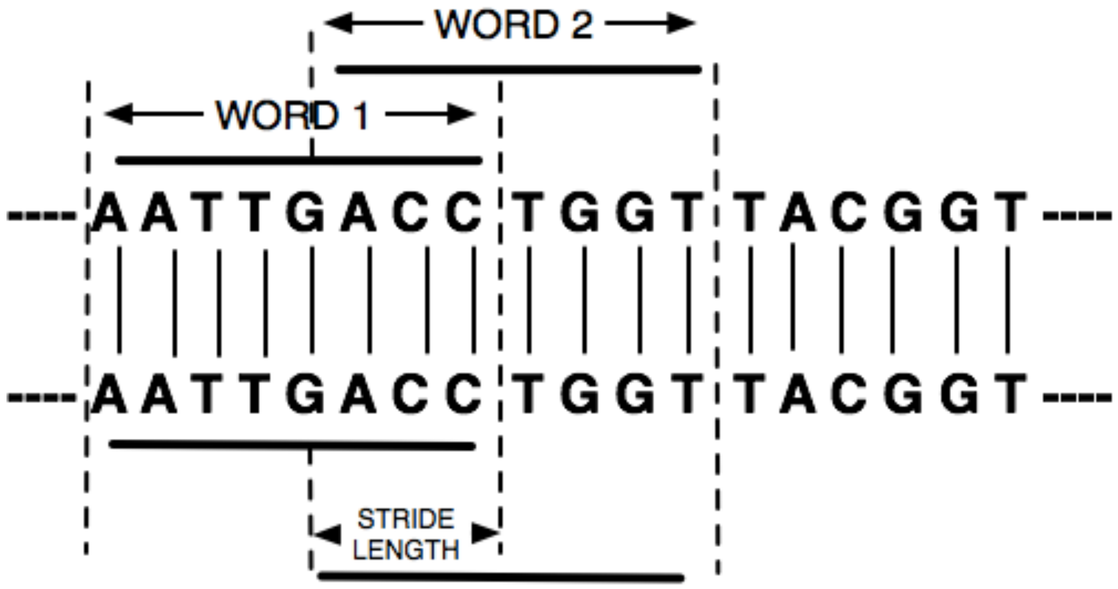


Figure 2.14: Example of word and stride length.

In more details, when the stride is 1 or 3 plus a multiple of 4, the scanning loop is unrolled by 4, and when it is 2 plus a multiple of 4 the scanning loop it is unrolled by 2. The unrolling process eliminates several shift and mask operations, and turns the shift and mask quantities into compile-time constants. When the scanning starts, the unrolled routines jump into the middle of the unrolled loop using a switch statement. In case the stride is less than 4, the scanning uses an accumulator to store words across scanning iterations. This reduces the number of memory accesses to the subject sequence, and the number of pointer increments. If the word size is a multiple of 4, mask operations are removed when the offset into the subject sequence is known to be a multiple of 4. Because the loops are unrolled, all strides benefit of this optimization. In the specific case in witch the lookup table width is 8 and the stride is 4, all the unrolled loop iterations are similar, that's why the scanning can be made to finish only at the bottom of the loop. This eliminates several bounds checks, and makes this case the fastest of all the scanning routines. Happily, this loop is also the most common for ordinary blastn.[11]

As far as our study concerns, the scanSubject routine is s_BlastSmallNaScanSubject_8_4, which belongs, in the last category mentioned above and a detailed flow chart is shown in figure 2.16. It uses as input a pointer to the look-up table, a pointer to the scan subject, an array of query and subject positions where word hits are found and which is also the output of our routine, a variable representing the maximum number of hits and the maximum length of the array mentioned above and the starting and ending position on the subject sequence being scanned. During its execution time two more routines are used, these are SMALL_NA_ACCESS_HITS and s_BlastSmallNaRetrieveHits. The first one is used to check if any hit exists and if so, it makes a call to the second one, which is responsible for their retrieval and their restoration in the right position of the output array. Detailed flow charts of both routines are shown in figures 2.17 and 2.18.

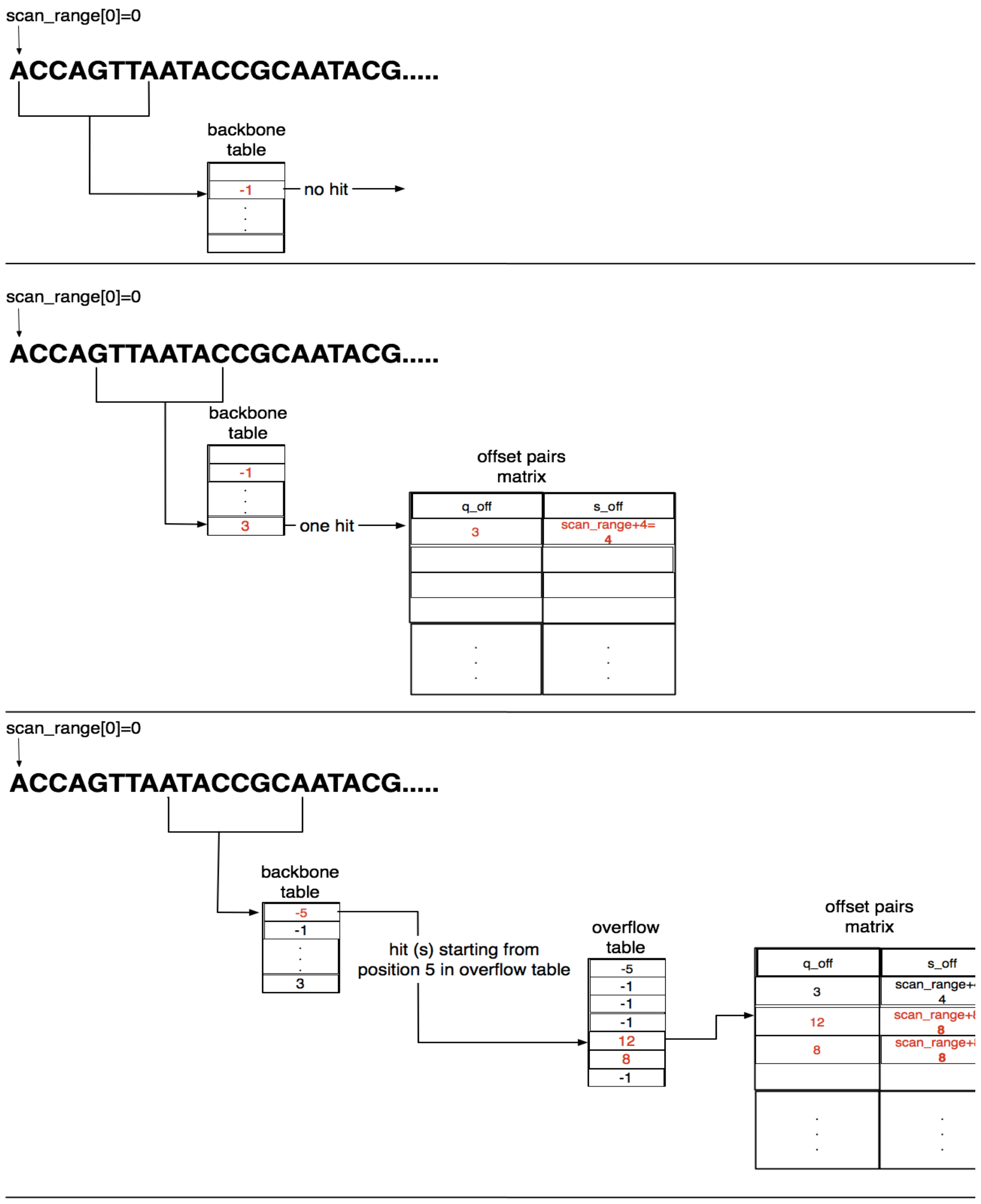


Figure 2.15: Seeding step using s_BlastSmallNaScanSubject_8_4.

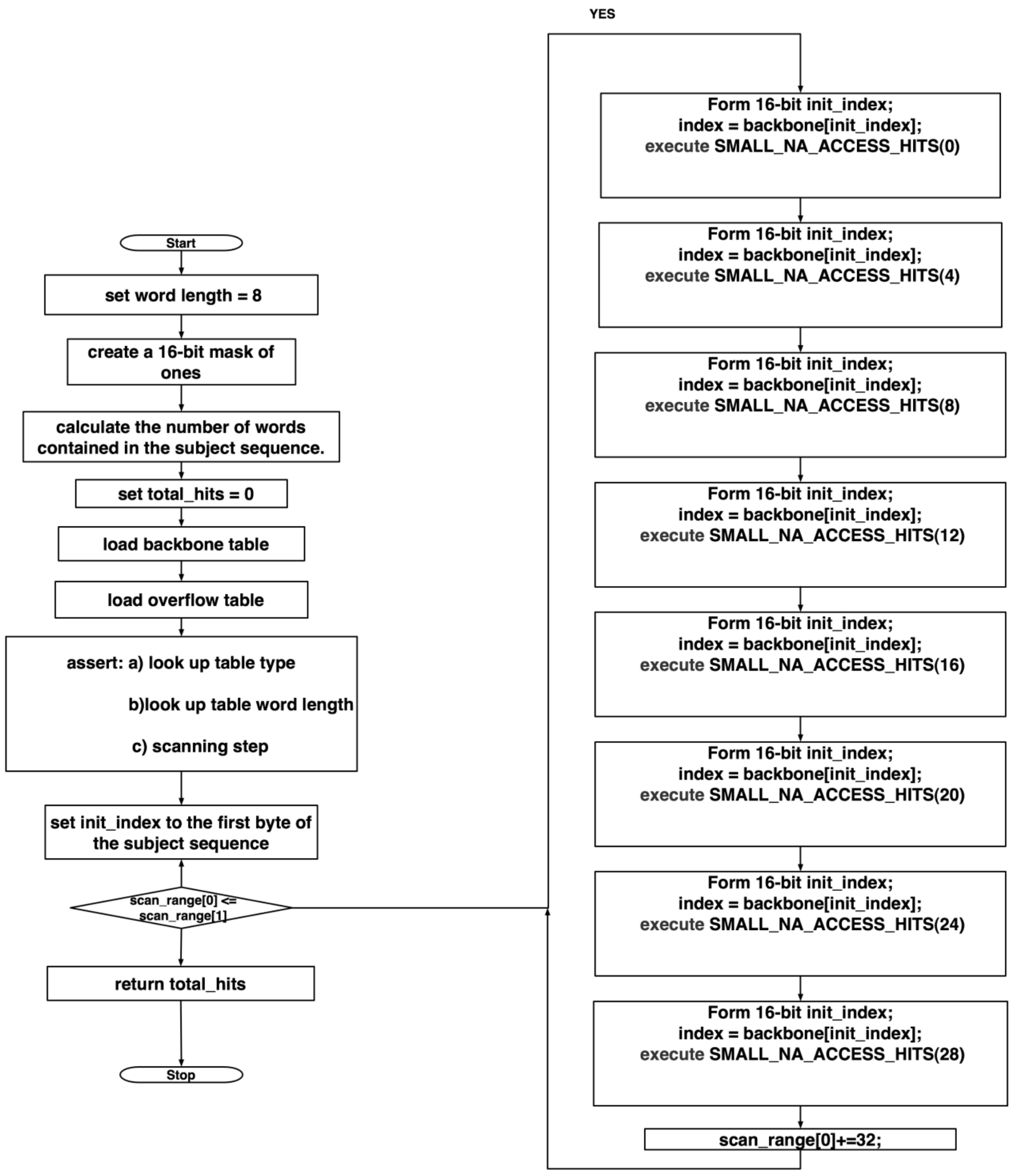

YES
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(0)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(4)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(8)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(12)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(16)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(20)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(24)
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(28)
scan_range[0]+=32;
Start
set word length = 8
create a 16-bit mask of ones
calculate the number of words contained in the subject sequence.
set total_hits = 0
load backbone table
load overflow table
assert: a) look up table type
b)look up table word length
c) scanning step
set init_index to the first byte of the subject sequence
scan_range[0] <= scan_range[1]
return total_hits
Stop

Figure 2.16: Detailed flow chart of s_BlastSmallNaScanSubject_8_4 scanSubject routine.

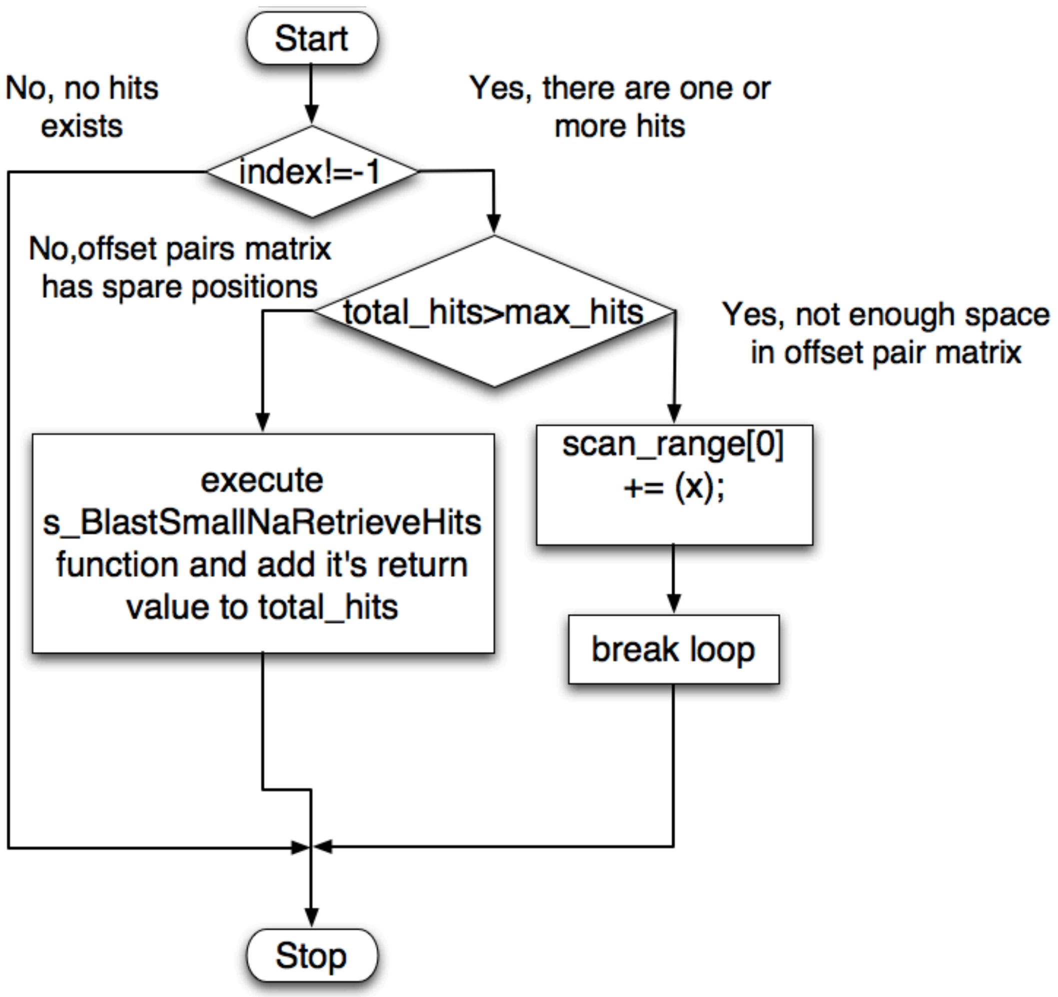

Start
No, no hits exists
Yes, there are one or more hits
index!=-1
No,offset pairs matrix has spare positions
total_hits>max_hits
Yes, not enough space in offset pair matrix
execute s_BlastSmallNaRetrieveHits function and add it's return value to total_hits
scan_range[0] += (x);
break loop
Stop

Figure 2.17: Flow chart of SMALL_NA_ACCESS_HITS routine, used by the seeding step of the software version of NCBI_BLASTn. It is called by s_BlastSmallNaScanSubject_8_4 routine.

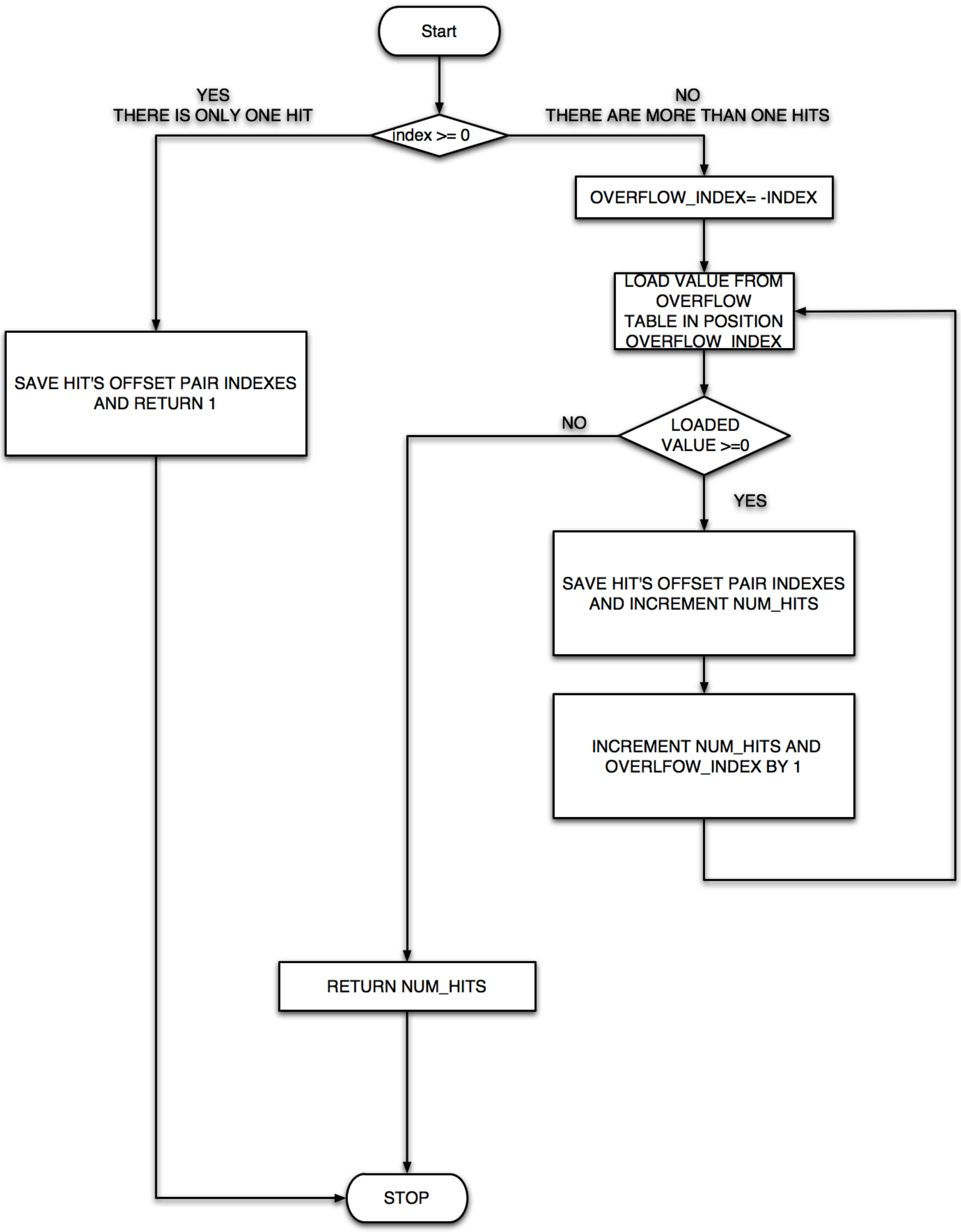

Start
YES
THERE IS ONLY ONE HIT
index >= 0
NO
THERE ARE MORE THAN ONE HITS
OVERFLOW_INDEX= -INDEX
LOAD VALUE FROM OVERFLOW TABLE IN POSITION OVERFLOW_INDEX
SAVE HIT'S OFFSET PAIR INDEXES AND RETURN 1
NO
LOADED VALUE >=0
YES
SAVE HIT'S OFFSET PAIR INDEXES AND INCREMENT NUM_HITS
INCREMENT NUM_HITS AND OVERLFOW_INDEX BY 1
RETURN NUM_HITS
STOP

Figure 2.18 : Flow chart of s_BlastSmallNaRetrieveHits routine, used by the seeding step of the software version of NCBI_BLASTn. It is called by SMALL_NA_ACCESS_HITS predefined routine.

2.6 Conclusions

This chapter analyzes BLAST algorithm and its basic steps. Also, this chapter analayzes the computational performance of the NCBI-BLAST software implementation and its calling routines.

There are some conclusions that come out from this chapter:

1. BLAST algorithm is the most well knonw algorithm for sequence alignment.

2. BlAST consists of three steps, which are seeding, extention and evaluation.

3. BLAST algorithm relies on deferent substitution matrices.

4. NCBI BLAST is a suite of programs for deferent kind of alignments.

5. NCBI-BLAST programs are : BLASTP, BLASTN, MEGABLAST, PSIBLAST,BLASTX,TBLASTX,RPSBLAST,TBLASTN and PHIBLAST.

6. BLAST is a very computationaly expensive algorithm. Its execution time depends besicaly on the input sequence and database.

7. The most computational intensive part of the algorithm is the seeding step and it takes up to 50 % of the execution that's why it was chosen to implemented on FPGA.

8. NCBI-BLASTn has about a dozen of functions to implement the seeding step. The function, which is suitable for small query implemented on FPGA.

3. Convey HC-1

Convey Computers made a revolution to the high-performance computing (HPC) field by launching the world's first hybrid-core computer HC-1 that breaks the barriers of expensive power, performance and programmability. The HC -1 server managed to change the till today known HPC as it breaks through the current power/performance wall to significantly increase performance for certain compute and memory bandwidth intensive applications. Also, it is easy for programmers to use as it provides full support of an ANSI standard C, C++ and Fortran development environment and it significantly reduces support, power and facility costs for companies.

Convey with HC-1 managed to fill a market space called hybrid-core computing, which marries low cost and simple programming model of a commodity system with the performance of customized hardware architecture.

3.1 HC-1 server

Convey's infrastructure combines an Intel Xeon processor and a Convey designed coprocessor based on Xilinx Field Programmable Gate Arrays, with its own high-bandwidth, virtual memory addressed, cache-coherent memory subsystem. It also offers an ANSI standard development environment, increasing productivity and portability.

HC-1's main strength is Convey's implementations, called Personalities, which are extensions to the x86 instruction set that are implemented in hardware increasing productivity and optimizing performance of specific portions of an application. They are sharing the same physical and virtual address spaces with the x86 instructions, and applications can contain both x86 and coprocessor instructions in a single-instruction stream. Convey compilers generate one executable image that contains both x86 and coprocessor instructions. Systems can contain multiple personalities. Convey provides a Personality Development Kit (PDK) for creation of new application-oriented architectures discussed in details later in this chapter.

Another strong point of HC-1 is its memory, which provides a bandwidth of 80 Gigabytes/sec delivering huge sustainable performance. A shared virtual and physical memory between the coprocessor and the x86 provide the tight integration that allows the system to be programmed as a single architecture. This means that the programmer does not need to manage the physical memory on the coprocessor nor explicitly move data back and forth between the x86 main memory and the coprocessor main memory.

3.2 Hc-1 system architecture

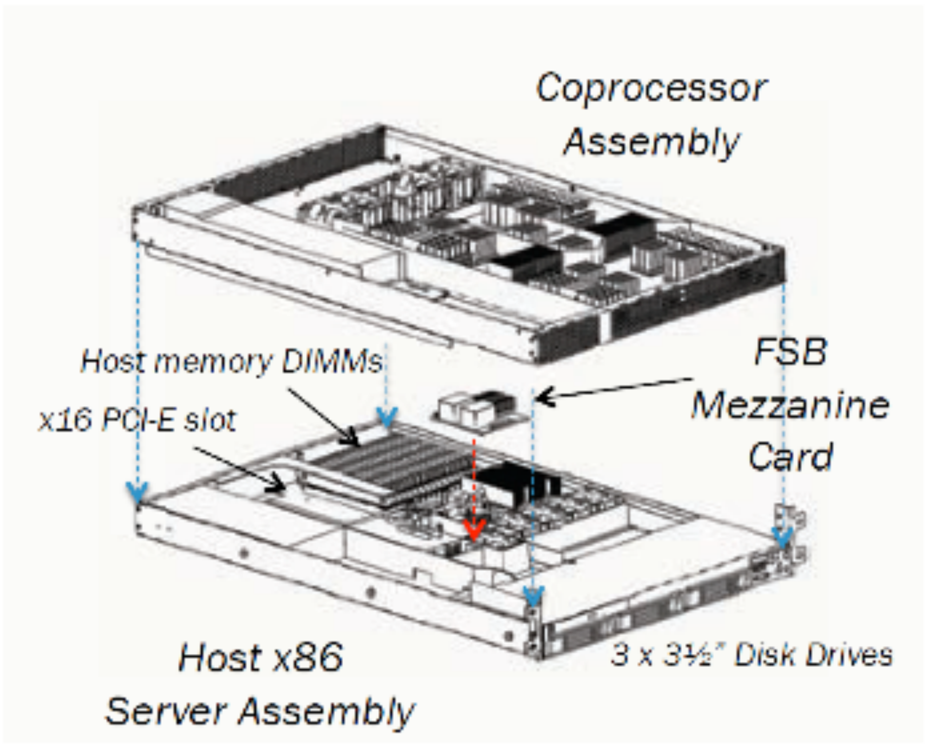


Figure 3.1 HC-1 server architecture.

Convey HC-1 is a hybrid-core computer system that uses a commodity two-socket motherboard to combine a reconfigurable, FPGA-based coprocessor with an industry standard Intel 64 host processor. Physically, the system is based on two main logic boards in a rack-mountable 2U enclosure. The top half of the enclosure is the coprocessor and the bottom half is the commodity motherboard. A mezzanine interconnection mechanism connects the halves and extends the host motherboard's front-side bus (FSB) to the coprocessor. The entire system consumes approximately 600 watts with the coprocessor executing code. The system architecture is shown in Figures 3.1 above and 3.2 below.

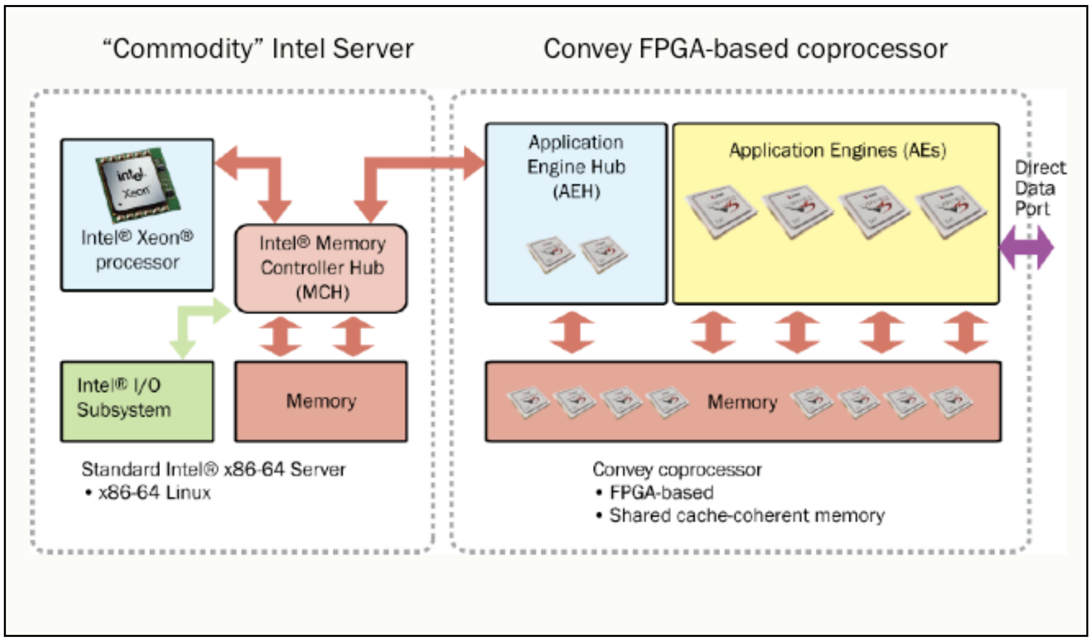

"Commodity" Intel Server
Convey FPGA-based coprocessor
Intel® Xeon® processor
Intel® Memory Controller Hub (MCH)
Intel® I/O Subsystem
Memory
Standard Intel® x86-64 Server
• x86-64 Linux
Application Engine Hub (AEH)
Application Engines (AEs)
Direct Data Port
Memory
Convey coprocessor
• FPGA-based
• Shared cache-coherent memory

Figure 3.2 HC-1 Architecture.

3.2.1 Intel host processor

HC-1's host consists of a dual socket Intel server motherboard an Intel 5400 memory-controller hub chipset, 1,066 MHz FSB and a 2,13 GHz dual core Intel Xeon low voltage processor. The HC-1 host runs a 64-bit 2.6.18 Linux kernel with a Convey modified virtual memory system for memory-coherent with the coprocessor board memory reasons.

3.2.2 Convey FPGA-based coprocessor

Convey's HC-1 coprocessor composed by three main sets of components. The Application Engines (AEs), the Memory Controllers (MCs), and the Aplication Engine Hub (AEH).

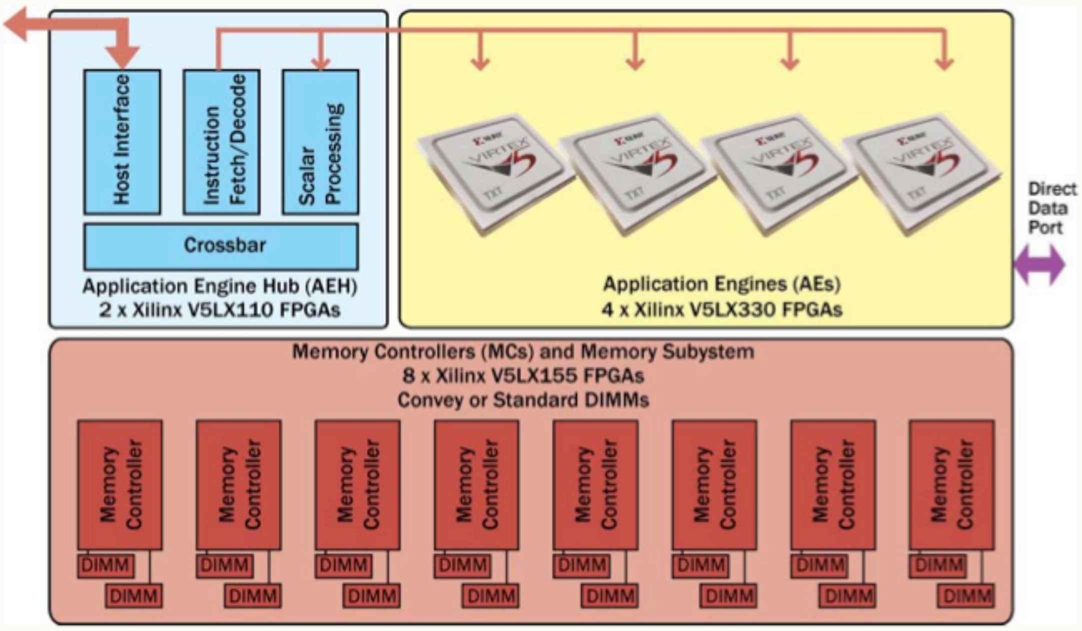


Figure 3.3 Coprocessor three main components AEH, AEs, MCs.

The Application Engine Hub (AEH) is the coprocessor's to host interface, it consists of two non–user programmable Xilinx V5LX110 FPGAs. One serves as the physical interface between the coprocessor board and the FSB, it monitors the FSB to maintain the snoopy memory coherence protocol and manages the coprocessor memory's page table. This FPGA is actually mounted to the mezzanine connector. The second one contains a soft-core scalar processor, which implements the base Convey instruction set. It is also the mechanism by which the host invokes computations on the AEs.

To support the bandwidth demands of the coprocessor, 8 Memory Controllers (MCs) are used. Each memory controller is implemented on its own FPGA and is connected to two standard DDR2 dual inline memory modules (DIMMs) or to two Convey-designed scatter-gather dual inline memory modules (SG-DIMMs), containing 64 banks each and an integrated Stratix-2 FPGA. The SG-DIMMs allow access to physical memory by quad words (8 bytes) instead of by 64-byte cache lines (as the host does). Accessing by 8-byte blocks reduces the inefficiencies encountered when accessing memory by no unity strides (or randomly) with a cache-based system. The inefficiency can be as drastic as one eighth of the peak bandwidth, because if only 4 or 8 bytes out of an entire 64-byte cache line are needed, the rest of the transfer is wasted.

The Application Engines (AEs) are four user-programmable Virtex-5 XC5VLX330 FPGAs, which are the heart of the coprocessor and implement the extended instructions that deliver performance for a "personality" which is a particular configuration of these FPGAs .The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands, and to the memory controllers via a network of point-to-point links that provide very high sustained bandwidth. Each AE instruction is passed to all four AEs. The way that they process the instructions depends on the personality. The AEs are interconnected with 668 Mbytes/s, full duplex links for AE to AE communication.

Each AE has a 2.5 GB/s link to each memory controller, and each SG-DIMM has a 5 GB/s link to its corresponding memory controller. The effective memory bandwidth of the AEs is dependent on their memory access pattern to the eight memory controllers and their two SG-DIMMs. Each AE can achieve a theoretical peak bandwidth of 20 Gbyte/s when striding across eight different memory controllers, but this bandwidth would drop if two other AEs attempt to read from the same set of SG-DIMMs because this would saturate the 5 Gbytes/s DIMM memory controller links [12].

The Convey memory system using Scatter/Gather DIMMs has 1024 memory banks. The banks are spread across eight memory controllers (MCs). Each memory controller has two 64-bit busses, and each bus in accessed as eight sub busses (8-bits per sub bus). Finally, each sub bus has eight banks. The 1024 banks is the product of 8 MCs * 2 DIMMs/MC * 8 sub bus/DIMM * 8 bank/sub bus. The coprocessor memory hierarchy is shown in figure 3.4.

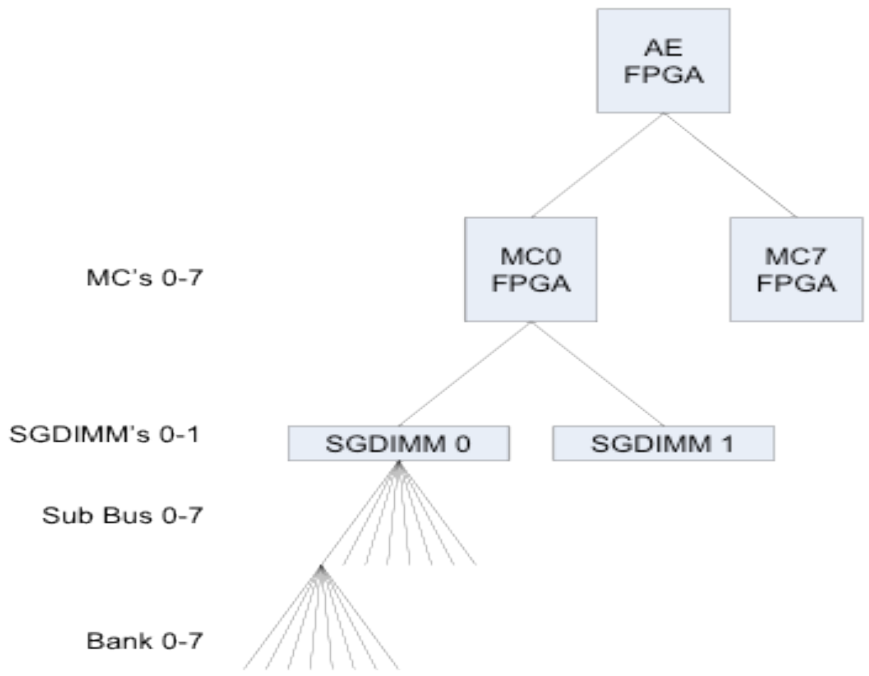


Figure 3.4 Coprocessor's memory hierarchy scheme.

Convey provides two user-selectable memory mapping modes to partition the coprocessor's virtual address space among the SG-DIMMs:

• **Binary interleave**, which maps bitfields of the memory address to a particular controller, DIMM, and bank.

• **31-31 interleave**, a modulo 31 mapping optimized for constant memory strides (strides lengths that are a power-of-two are guaranteed to hit all 16 SG-DIMMs for any sequence of 16 consecutive references).
The memory banks are divided into 32 groups of 32 banks each. In 31-31 interleave, one group isn't used, and one bank within each of the remaining groups isn't used. Because the number of groups and banks per group is a prime number, this reduces the likelihood of strides aliasing to the same SG-DIMM. The operating system also supports page coloring such that every time a page is allocated from coprocessor memory, it's allocated at a specific physical address to maintain the prime number interleaving. Selecting the 31-31 interleave comes at a cost of approximately 1 Gbyte of addressable memory space (6 %) and a 6 percent reduction in peak memory bandwidth.

The 31/31 interleave scheme was defined to meet the following requirements:

1. Provide the highest possible bandwidth for all memory access strides, with a particular focus on power of two strides.

2. Keep each memory line (64-bytes) on a single memory controller. This is required to simplify the cache coherency protocol.

3. Maintain the interleave pattern across virtual memory page crossings. This helps large strides where only a few accesses are to each page.

4. All virtual addresses must map to unique physical addresses.[12]

When Binary interleave is enabled the 1024 banks are accessed with the following virtual address assignment:

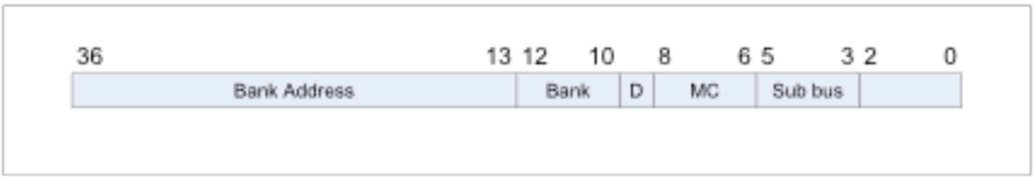


Figure 3.5 Access pattern of Virtual address.

In this Figure 3.5 is the targeted DIMM, and MC is the memory controller. In practice, the hardware design should check bit 8:6 of the virtual address and send the request to the appropriate MC. To be able to run at (near) peak bandwidth, all the requests must be equally distributed over the MCs, but also over the banks and sub busses.

The memory system also supports large (4 Mbyte) virtual pages, allowing the entire physical address space to be mapped into the page translation look aside buffers (TLB) within the memory controllers. By doing so, an application will only have to fault in a page once for the application's entire execution, which eliminates TLB thrashing. Further, each TLB entry contains a process identifier, which allows TLB entries to survive process exchanges (if not overwritten).

The coprocessor memory is cache coherent with the host memory by using the snoopy coherence mechanism built into the Intel FSB protocol. A virtual address space is created that both the host and coprocessor share. In the coherence protocol, both the host and the coprocessor possess copies of the global memory space. Each block of memory addresses in both the host memory and coprocessor memory are marked as exclusive, shared, or invalid. A write by the host to an address block will change its status to exclusive and invalidate the block on the coprocessor (indicating that it's out-of-date). If one of the application engines on the coprocessor reads from this block, an updated copy of the block's memory contents is sent to the coprocessor memory, and the memory block changes to shared in both the host and coprocessor memory. The coherence mechanism is transparent to the user and removes the need for explicit direct memory access (DMA) transactions, which coprocessors based on peripheral component interconnect (PCI) require [13].

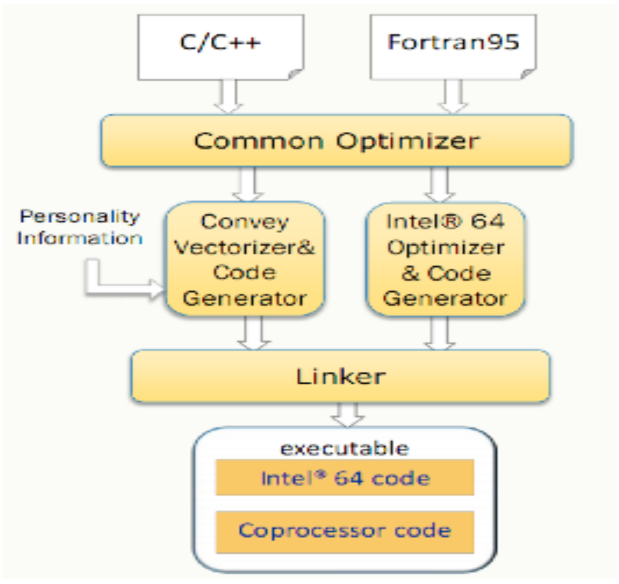


Figure 3.6 HC-1's Programming model.

3.3 Programming model

In Convey's programming model applications can be coded in standard C, C++, or Fortran, the AEs act as co-processors to the scalar processor, while the scalar processor acts as a co-processor to the host CPU. Because of this, the executable file on the host contains integrated scalar processor code. This is transferred to and executed on the scalar processor when the host code calls a scalar processor routine through one of Convey's runtime library. The scalar processor code can contain instructions that are dispatched and executed on the AEs. The final executable is generated by a unified compiler and it integrates both x86 and co-processor instructions defined by the personality used on compilation time. [14]. Figure 3.6 shows an abstract view of the programming

3.4 Debugging

When it comes to debugging Convey provides the necessary environment for host and co-processor code debug. This environment is composed by a version of the GDB debugger that has been extended with coprocessor-specific register state, for debugging x-86 and coprocessor routines. A dedicated PCIe link to the Management Processor (MP) on the co-processor board, independent of the application data-path link through FSB that configures and monitors the FPGAs, and it also provides visibility, when the AE FPGAs are in bad state and appear hung. Despite Convey providing Control and Status Register (CSR) agents to get visibility into the FPGA, the Chipscope logic analyzer tool can also be used [14].

3.5 Convey Simulator and Performance Analysis Tool

To help developers get the best performance out of their code, Convey also offers a simulator and corresponding performance analysis tool called "Spat" [15] that graphically plots how various aspects of the code map to the architecture and can assist in code tuning. Information is presented as a plot of clock cycle versus usage of various architectural features. The tool can also graphically depict detailed state information for various units within the scalar and vector processors. This gives the ability to users to step across clock cycles and watch how the system executes various instructions [13].

3.6 Porting already existing applications

Convey offers four different ways for porting already existing applications to their HC-1 system mentioned below:

- Use the Convey Mathematical Libraries (CML), which is a set of functions optimized for the co-processor, which use predefined Convey-supplied personalities, for example Convey's CML-based FFT uses the single-precision personality.

- Compile one or more routines with Convey's compiler. This uses the Convey auto-vectorization tool to automatically select and vectorize do/for loops for execution on the co-processor. Directives and pragmas can also be manually inserted in the source code, to explicitly indicate which part of a routine should execute on the co-processor.

- Develop hand-code routines in assembly language, using both standard instructions and personality specific accelerator instructions. Which can be called from C, C++, or Fortran code.

- Develop a custom personality using Convey's Personality Development Kit (PDK), to give the ultimate in performance using a hardware description language such as Verilog or VHDL [14].

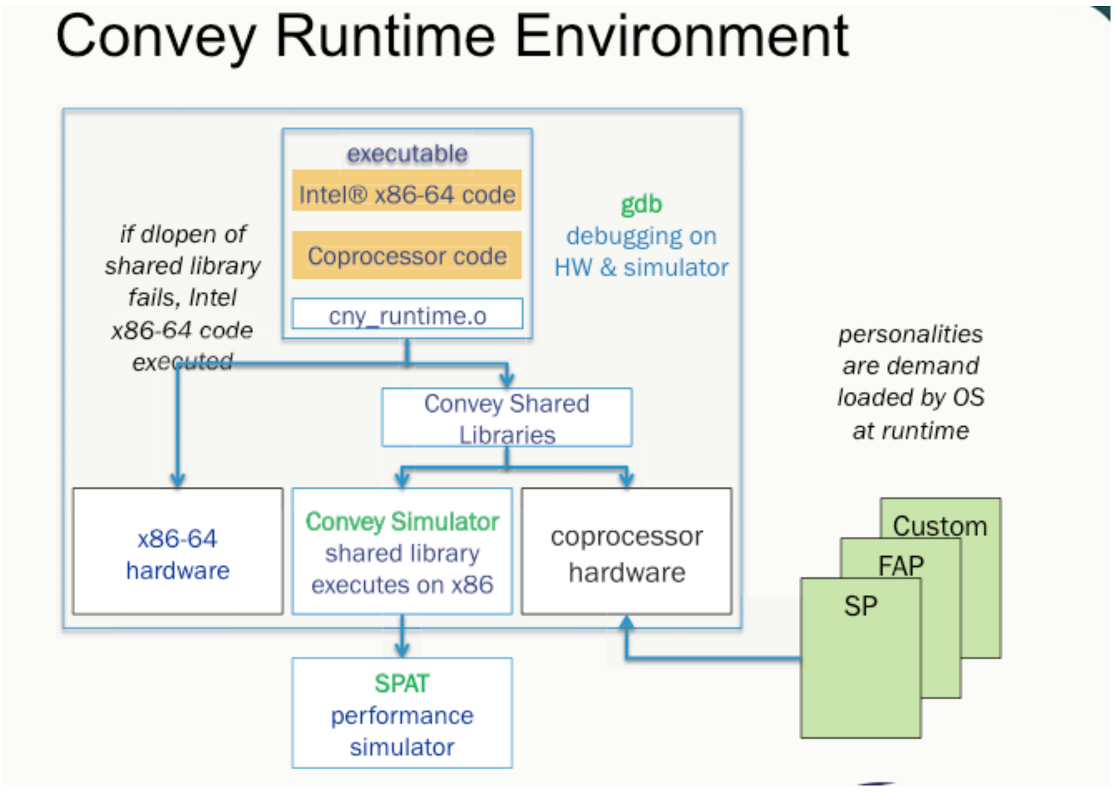

Convey Runtime Environment
executable
Intel® x86-64 code
Coprocessor code
cny_runtime.o
gdb
debugging on HW & simulator
if dlopen of shared library fails, Intel x86-64 code executed
Convey Shared Libraries
personalities are demand loaded by OS at runtime
x86-64 hardware
Convey Simulator shared library executes on x86
coprocessor hardware
Custom
FAP
SP
SPAT performance simulator

Figure 3.7 HC-1's runtime environment.

3.7 Personalities

Personalities are the key to the Convey systems' performance and flexibility. A personality includes the precompiled FPGA bit files that implement a coprocessor instruction set, a description of the machine state model sufficient for the compiler to generate and schedule instructions, and an ID used by the application to load the correct image at runtime. A system can contain multiple personalities that can be dynamically loaded, but only one personality is loaded at any one time. Each personality supports the entire canonical instruction set, plus extended instructions that may be unique to that personality. Extended instructions are designed for particular workloads, and may include only the operations that represent the largest portion of the execution time for an application.
All personalities have some elements in common, however:

- Coprocessor execution is initiated and controlled via instructions, as defined by the Convey Instruction Set Architecture.

- All personalities use a common host interface to dispatch coprocessor instructions and return status. This interface uses shared memory and leverages the cache coherency protocol to minimize latency.

- Coprocessor instructions use virtual addresses and coherently share memory with the host processor. The host processor and I/O system can access coprocessor memory and the coprocessor can access host memory. The virtual memory implementation provides protection for process address spaces as in a conventional system.

- All personalities support the canonical instruction set, and the Convey compilers assume that the canonical instructions can be generated and executed.

These common elements ensure that compilers and other tools can be leveraged across multiple personalities, while still allowing customization for different workloads.

Convey has developed the ability to swap personalities. For example, a user has a code that processes large arrays that can be easily vectorized followed by a series of long state machines. When the application lands on the first vector instruction, the AEH send a signal to the Application Engines to reconfigure themselves into a wide vector processor. At the point where the vector instructions are complete, the AEH signals again to switch to a specialized state machine. This hardware context switching occurs in real time with very little latency to the application. The AEH also caches these in the event that they must be reused.

Convey develops and licenses its own set of personalities but also allows users to design and implement their own custom personality using the personality development kit (PDK).

3.7.1 Convey provided personalities

Convey's set of personalities includes a single-precision vector personality and a double-precision vector personality which are vector coprocessors for the scalar processor and are targets for Convey's vectorizing compiler. For these personalities, each AE implements eight floating point multiply adder pipelines and eight load/store units (for a total of 32 logically combined across four AEs). Financial analytics personality, which is a double-precision personality that adds additional vector instructions, transcendental functions, probability distribution functions, and various random number generators designed for high-performance Monte Carlo simulation, is also included in Convey's set. In addition Convey provides several applications paired with a custom personality. These applications are based on existing 3-rd party applications, and can provide huge speedups while producing identical or very similar results, for the Bio-informatics industry these are:

- **Convey Graph Constructor**

  Generates simplified de Bruijn graphs from short read sequence data generated by modern sequencers. It reduces the execution time and memory required for graph construction.

- **Convey Sequencing Library**

  A simple, reusable interface to common algorithmic constructs utilized for sequencing and alignment operations.

- **Blast applications and associated personality (cnyBLASTp and cnyBLASTx)**

  An extension of NCBI BLAST which interfaces with an FPGA based pre-filter to provide accelerated performance of protein-based searches on Convey hybrid-core servers.

- **Convey Smith-Waterman Search application and associated personality**

  A protein sequence search program using the Smith-Waterman algorithm, optimized for the Convey hybrid-core architecture.

As mentioned earlier, users who wish to develop their own custom personalities with HDL-based design or C-to-HDL third party tools for accelerating personality Development must license the PDK, which includes design flows and robust system models that support hardware/software co-simulation. A custom personality can be developed for many applications, to utilize the full potential of the Convey coprocessor. By developing a personality for a specific application, the Convey coprocessor can make the most effective use of the FPGAs of the coprocessor. A custom personality implements a number of functional units within the coprocessor, each of witch can perform the same calculations on different data, in parallel with the other functional units.

3.8 The Convey Personality Development Kit

The Personality Development Kit is a set of tools and infrastructure that enables development of a custom personality for the Convey HC-1 system. A set of generic instructions and defined machine state in the Convey instruction-set architecture allows the user to define the behavior of the personality. Logic libraries included in the PDK provide the interfaces to the scalar processor, memory controllers, to the inter-FPGA links and to the management processor for debug. We will present each of these interfaces with more detail in next session. The user develops custom logic that connects to these interfaces.

The Convey PDK provides the following set of features as a part of the kit:

- Makefiles to support simulation and synthesis design flows,

- A Programming-Language Interface (PLI) to let the host code interface with a behavioral HDL simulator such as Modelsim or Synopsys.

- FPGA hardware interfaces provided as Verilog modules, these interfaces connect custom personality hardware to instruction dispatch, management and memory resources on the coprocessor.

- Custom personality software and hardware simulation environment Bus -functional models are provided to connect each of the hardware interfaces to Convey's architecture simulator.

- A sample personality illustrates how to use the hardware and simulation interfaces to develop a custom personality.

In addition to the PDK package, several Convey software packages are required for PDK development. Which are:

- Xilinx ISE Design Software for synthesis, place and route of FPGAs.
- An HDL simulator for Verilog/VHDL simulation. Mentor ModelSim or Synopsys VCS. [12]

The PDK's simulation framework is easy to use and allows users to switch between a simulated coprocessor mode and an actual coprocessor, by changing a single environment variable.

3.8.1 Personality development steps

Convey provides ten step-by-step instructions for the process of developing a custom personality for its Coprocessor.

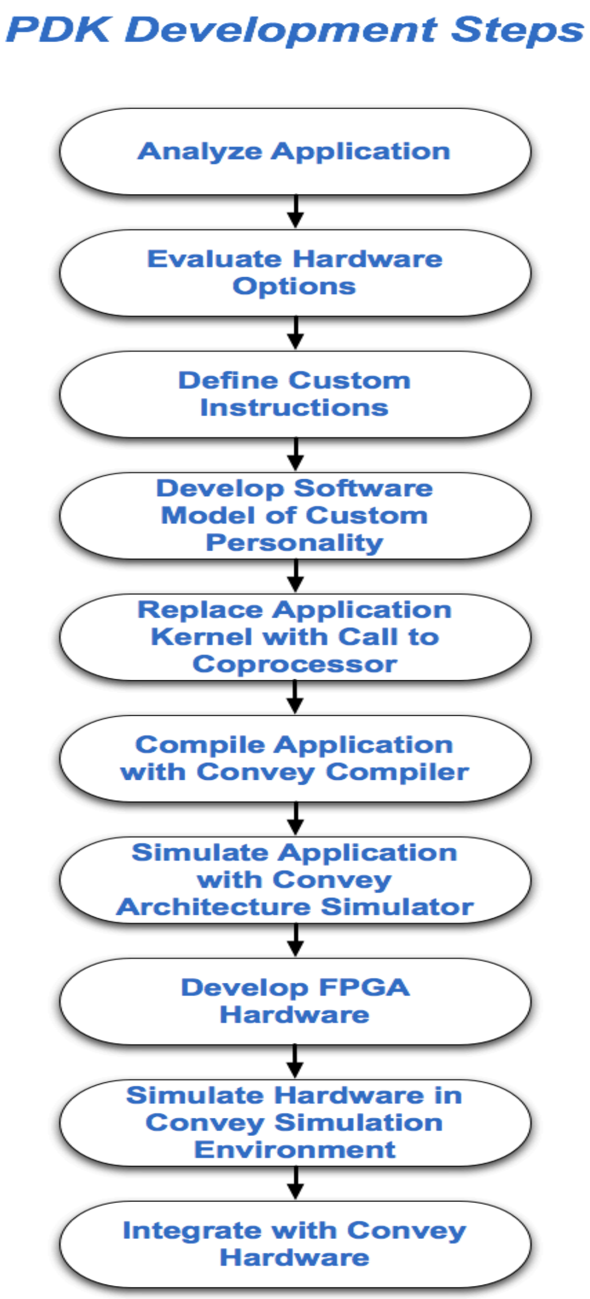

PDK Development Steps
Analyze Application
Evaluate Hardware Options
Define Custom Instructions
Develop Software Model of Custom Personality
Replace Application Kernel with Call to Coprocessor
Compile Application with Convey Compiler
Simulate Application with Convey Architecture Simulator
Develop FPGA Hardware
Simulate Hardware in Convey Simulation Environment
Integrate with Convey Hardware

Figure 3.8 PDK Development Steps proposed by Convey

1.Analyze Application

The first step of personality development is to completely understand the problem to be solved. How does the current application perform on existing hardware? What are the bottlenecks that limit the performance? What data structures are involved? How parallelizable is the application? Answers to these questions provide the first insight into how the application can be accelerated in hardware. Some tools that are useful in gathering this information are **gprof** (The GNU Profiler) and **oprofile**.

2.Evaluate Hardware Options

With a detailed knowledge of the application and its performance limitations,

the second step is to evaluate options for implementing the application in hardware. This requires a good understanding of the hardware architecture and the FPGA resources available to the custom personality.
Once a concept for hardware design is completed, the performance of the hardware can be compared vs. to the existing application performance.

3.Define Custom Instructions

With a hardware concept in place, the functions that are implemented by the hardware design can be mapped to custom instructions. These are a set of instructions in the Convey Instruction Set Architecture reserved for custom personalities. Instruction sets designed to be used across a wide variety of applications typically have a large number of instructions that perform relatively simple operations. As a custom personality is designed to improve the performance of a single application, it might implement very few instructions with much more complex behavior.

4.Develop Software Model of Custom Personality

Convey provides an architecture simulation environment to allow rapid prototyping of both the hardware and software components of a custom personality. This environment is written in C++ to emulate the rest of the system. It includes hardware models of instruction dispatch, register state and the memory subsystem.
With a hardware design concept in place, and a definition of custom instructions to interface to that hardware, a software model can be developed to emulate the hardware. The hardware model can then be simulated with the rest of the system to prove the concept before detailed design begins.

5.Replace Application Kernel with Call to Coprocessor

The application should be modified so that the application kernel can be called as a function. To dispatch instructions to the coprocessor, the kernel function call is replaced with a call to dispatch the function to the coprocessor. This function explicitly defines the custom instructions to be dispatched to the Application Engines. The sample application described later in this document illustrates the use of this interface.

6.Compile Application with Convey Compiler

The PDK package includes Convey64 compilers that can be used to compile coprocessor applications with direct calls to coprocessor functions.

7.Simulate Application with Convey Architecture Simulator

Once the AE software model is in place and the appropriate changes to the application have been made, the application can run against the Convey architecture simulator. This step allows the application and the custom instruction set to be debugged before the hardware is designed.

8.Develop FPGA Hardware

With an instruction set architecture defined, the hardware implementation can begin.

9.Simulate Hardware in Convey Simulation Environment

As an extension of the Convey architecture simulator, Convey provides a hardware simulation environment with bus-functional models for all hardware interfaces to the Application Engine (AE) FPGA. Using a standard VPI interface (Verilog Procedural Interface) the architecture simulator can be used to provide stimulus to the HDL simulation.

10.Integrate with Convey Hardware

The final step is to run the application on the Convey Coprocessor hardware.

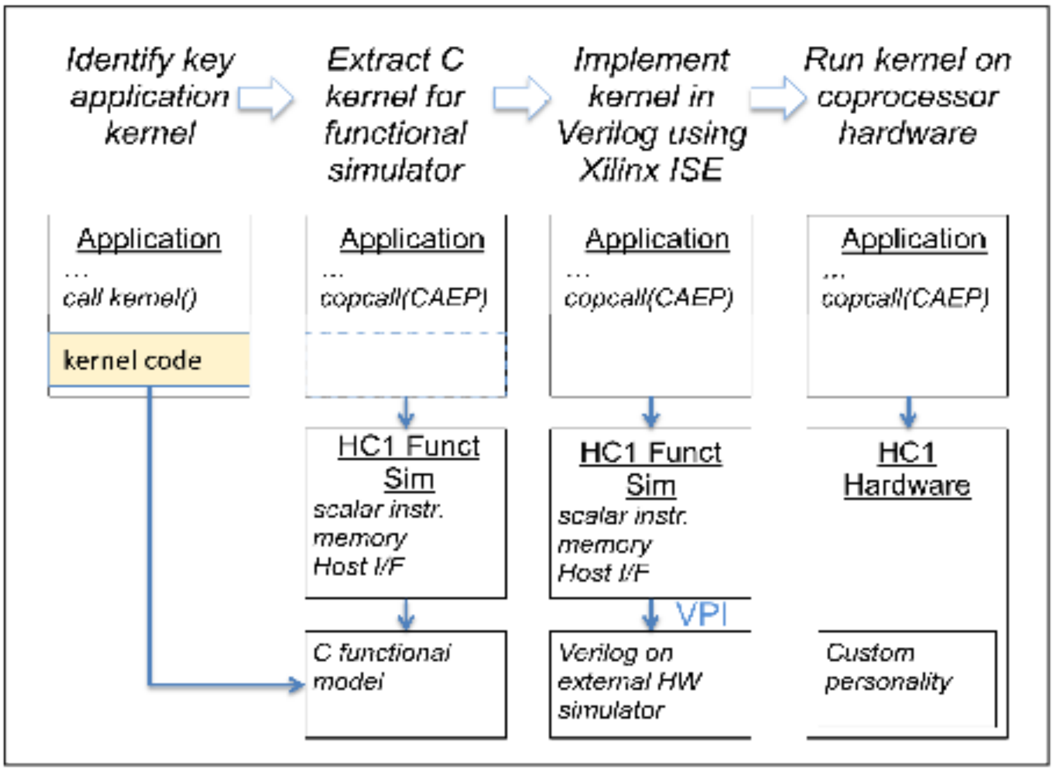


Figure 3.9 PDK design flow. [12]

3.9 Convey provided simulator

Convey provides a very useful Coprocessor architecture simulator for functional testing which contains a VPI (Verilog Procedural Interface) interface to an HDL simulator. This allows the actual user application, running on the architecture simulator, to provide the AE instructions for the hardware simulation of the FPGA.

The architecture simulator was developed to allow software to be tested and debugged in the absence of the actual Convey coprocessor hardware. It can also be used to prototype a PDK design quickly before investing the time to develop the FPGA hardware. The simulator models the machine state and the canonical instruction set, as well as instructions for the single and double-precision vector personalities designed by Convey. For custom personalities, the instructions set extensions are defined by the user and therefore cannot be modeled in the simulator, that’s why a socket interface is designed into the simulator to allow a user-developed AE software model to connect to the simulation process and emulate it. The application executable is a Linux executable, where host code calls to coprocessor routines are routed to the simulator. The host x86-64 code and the coprocessor simulator share the memory space of the executable, just as the real Convey coprocessor shares memory with the x86-64 host code.

The user’s developed software model must include implementation of the functions implemented and functions callable by custom personality. Such functions are personality initialization, modeling the instruction dispatch hardware interface where instruction decoding takes place, and memory and registers loads and stores. It also can be used as a checker in the hardware simulation.

Hardware simulation process (shown in the diagram below) needs an HDL simulator (ModelSim or VCS), which compiles the Verilog source for the Convey interfaces and custom logic, as well as the software model of the AE.

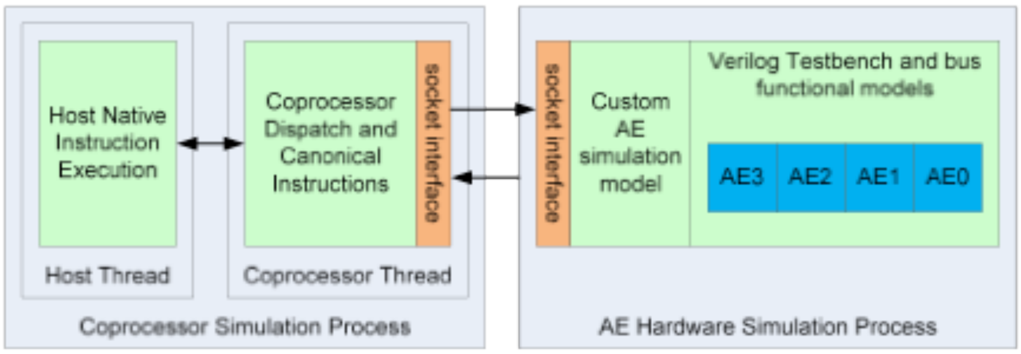

Host Native Instruction Execution
Coprocessor Dispatch and Canonical Instructions
socket interface
socket interface
Custom AE simulation model
Verilog Testbench and bus functional models
AE3
AE2
AE1
AE0
Host Thread
Coprocessor Thread
Coprocessor Simulation Process
AE Hardware Simulation Process

Figure 3.10 Custom AE hardware simulation.

During hardware simulation, the device under test (DUT) is the entire FPGA, consisting of the user-developed personality as well as the Convey-supplied hardware interfaces. The FPGA is instantiated in the testbench along with Verilog drivers and monitors for the FPGA interfaces. The bus functional models connect to the C-code portion of the simulation environment through

VPI.

3.10 PDK interfaces

For simple integration of a custom personality into the Convey coprocessor, PDK includes hardware interfaces for the Convey provided Verilog modules. Together with the custom personality module(s) developed by the user, these modules make up the design that will be synthesized into the AE FPGAs. These interfaces are:

3.10.1 Dispatch Interface

The dispatch interface is the hardware interface through which a host application sends coprocessor instructions to be executed by the AE. The dispatch interface receives instructions from the scalar processor. Some instructions are handled directly in the dispatch module. The dispatch interface also ensures that scalar data is returned to the scalar processor when required by the instruction.

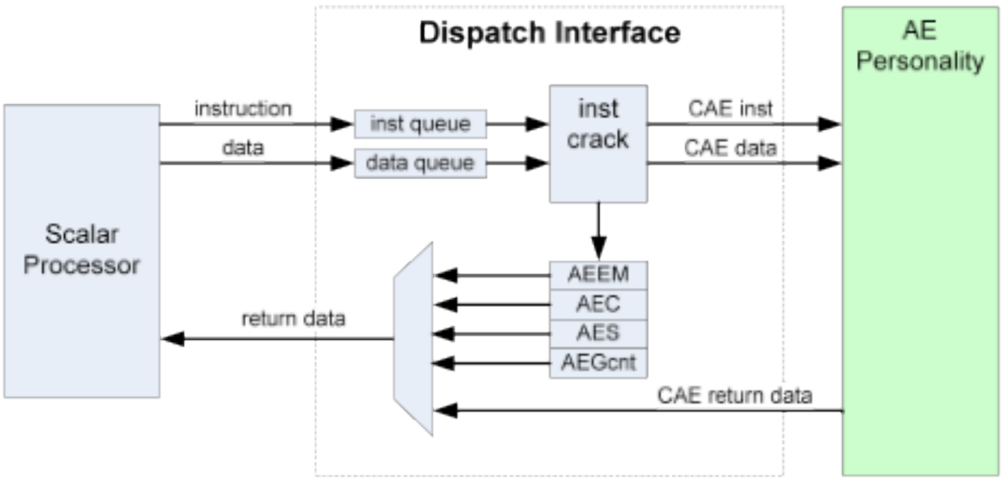


Figure 3.11 Dispatch Interface Diagram.

3.10.2 Memory Controller Interface

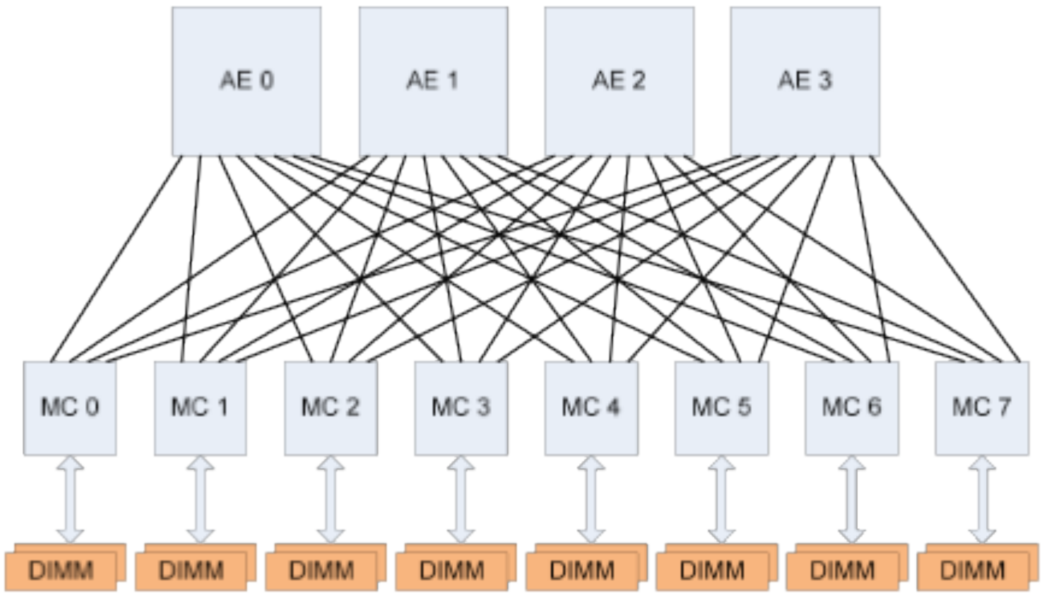


Figure 3.12 Coprocessor AE Memory Connections.

The Memory Controller (MC) Interface gives the AEs direct access to coprocessor memory. Each of the 4 AEs is connected to each of the 8 MCs (Memory Controllers) through a 300MHz DDR interface. The MC interface inside the AE FPGAs is provided by Convey. Each of 8 MC interfaces in the AE FPGA is directly connected to a single Memory Controller, and each MC physically connects to 1/8 of the coprocessor memory. The 8 MC interfaces are physically located on the left and right sides of the AE FPGA, as shown in figure 3.13 below. Each Memory Controller is connected to 2 DIMMs. The AE personality must decode the virtual memory address so that only requests intended for a particular MC's attached memory are sent to that MC.

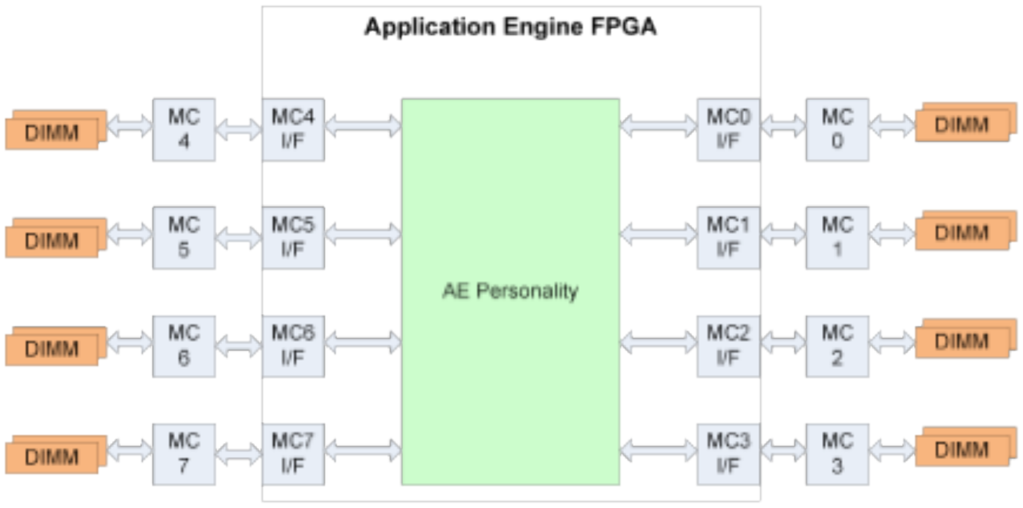


Figure 3.13 AE to MC Interface Diagram

The link between the AE FPGA and the MC FPGA runs at 300 MHz, but in order to ease timing in the FPGA, the 300 MHz interface is converted into two 150 MHz memory ports to/from the AE personality. Data from these two ports, the “even” port and the “odd” port, are multiplexed onto the same 300 MHz request channel in the MC interface.

For write operations, the write data is stored in a first-in, first-out buffer until it is sent across the AE-MC link. No response is returned to the AE personality for write operations.

For read operations, the write data bus is used to store read return control information. This data is stored in the write data buffer until the read request is sent out. When the read request is sent to the MC, the read request data is removed from the write data buffer and stored in a read control buffer based on the transaction ID assigned to the request transaction. When the read is returned from the MC, the transaction ID (TID) is used to lookup the read control information from the read control buffer.

The 32-bit read response control bus can be used by the custom personality for tracking request/response pairs. The data that is returned on this bus is the data that was written into bits <31:0> of write data when the read request was sent to the MC interface. Figure 3.14 below illustrates the functionality of the interface.

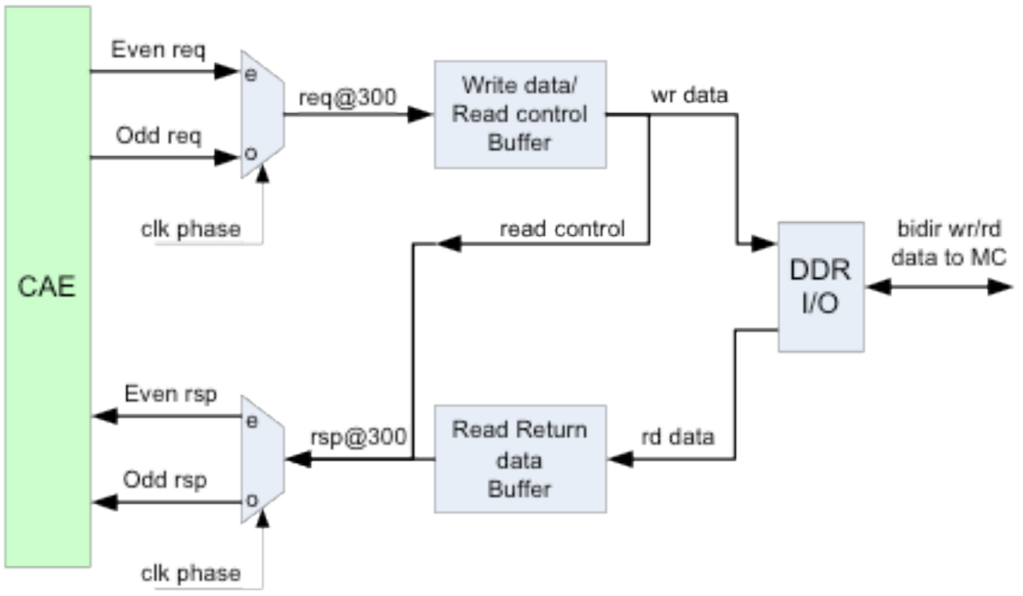


Figure 3.14 MC Interface Functional Block Diagram

3.10.3 AE-AE Interface

The AE-to-AE interface allows data to be transferred directly from one AE to another. Because the use of an AE-AE interface is unique to each application, it is difficult to design a solution that would be ideal for all custom personalities. Convey provides an AE- AE interface that the user may choose to use. The user is also free to use the signals between AEs in whatever way best supports their application. The Convey provided the AE-to-AE interface, which is designed with unidirectional busses to and from the previous or next AE. Each instance of the interface connects to a single AE, to connect an AE to both the previous and next AEs, two interfaces must be instantiated. This interface is simple and generic so that it can be used by many applications. Figure 3.15 below shows the AE-AE interface to the Custom Personality.

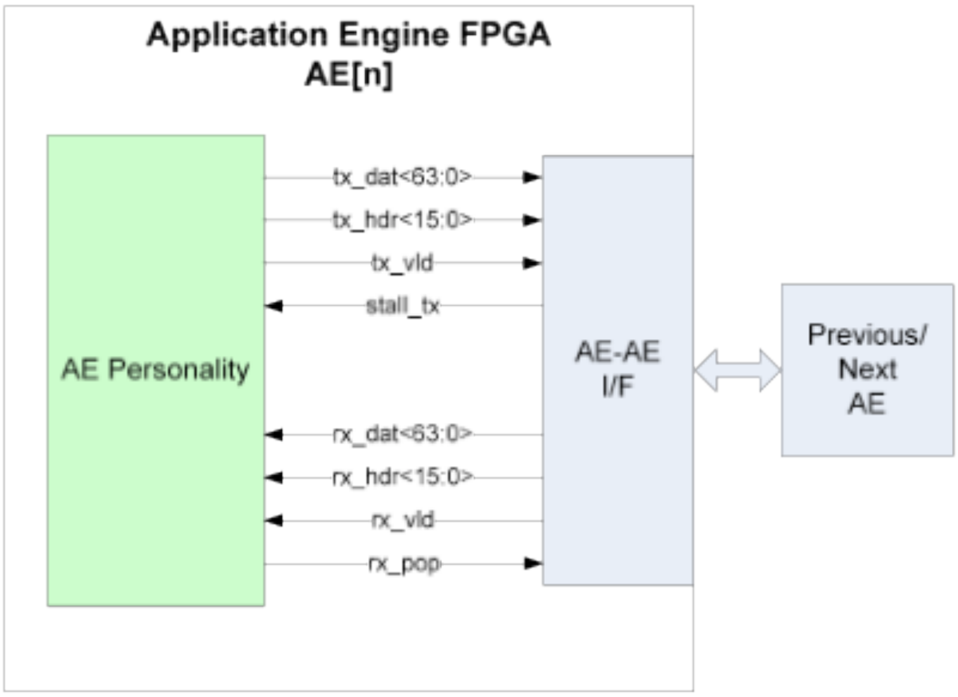


Figure 3.15 AE-to-AE Interface Diagram

3.10.4 Management/Debug Interface

The management interface provides the communication path between the Management Processor and the AE. The Management Processor (MP) is responsible for initialization and monitoring of the FPGAs. Since this path is independent of the instruction dispatch path from the host processor, it can be useful in debugging by allowing visibility into internal FPGA state, even when the application is hung.
The MP interface is instantiated in the Convey-supplied libraries, along with CSR agents in a ring topology. The custom personality must complete the ring by either adding one or more CSR agents to the ring or by simply connecting the inputs to the outputs. For many designs, a single agent is sufficient. For more complicated designs, the developer may choose to instantiate multiple CSR agents. The ring topology allows multiple agents to be placed near their associated logic. PDK CSR Registers are accessed from the host for debugging reasons. The host communicates with the MP FPGA via telnet.
Figure 3.16 below shows the connectivity of the CSR interface:

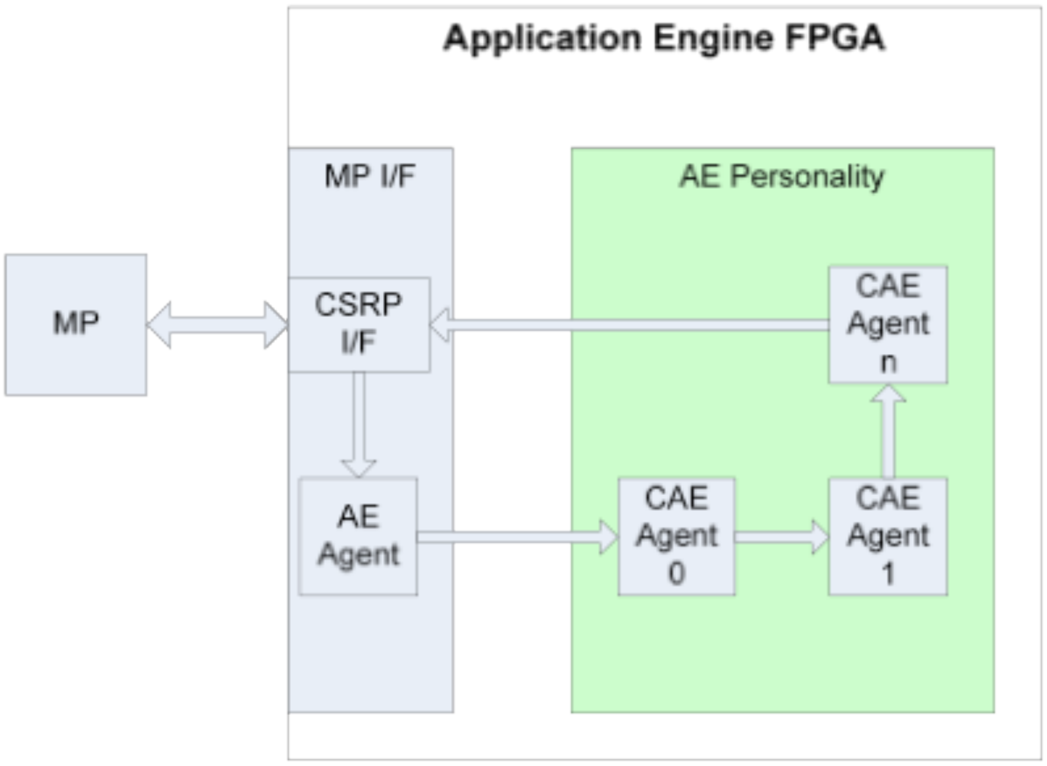


Figure 3.16 Management Interface Diagram

The Application Engines in the HC-1 platform are implemented in Xilinx Virtex 5 LX330 FPGAs. The required Convey hardware interfaces—the dispatch interface, CSR interfaces and MC interfaces—use about 10% of the available logic resources and about 25% of the block rams.

The dispatch interface occupies 400 slices in the center of the chip. The MC interfaces are on the left and right sides of the FPGA, and the MC CSR interfaces are in the corners of the part. Figure 3.17 below shows the FPGA floor plan.

There are 288 36Kb block RAMs in the LX330 FPGA. The required Convey interfaces use 66, leaving a total of 222 available for use by the custom personality. However, of the 222 available, 26 are on the left and right sides of the FPGA, which contain mostly MC interface and MC CSR logic. These block RAMs may or may not be useable, depending on the application and the timing requirements.

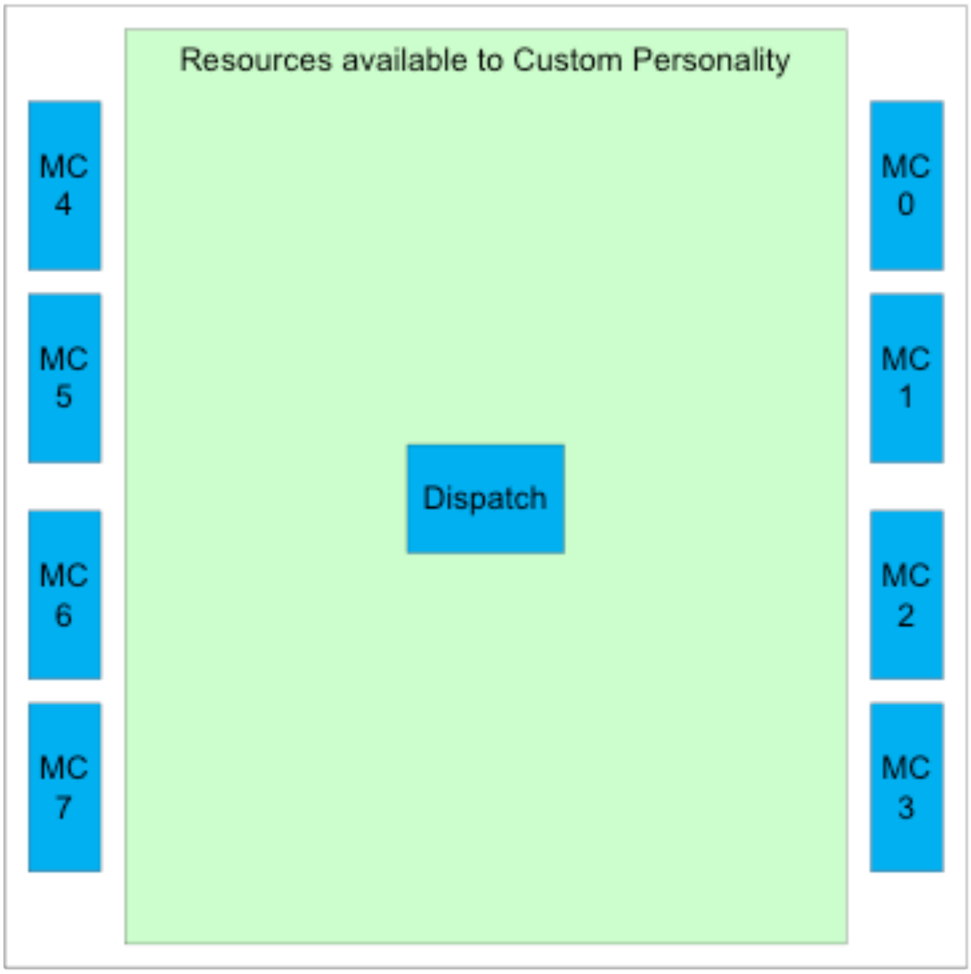


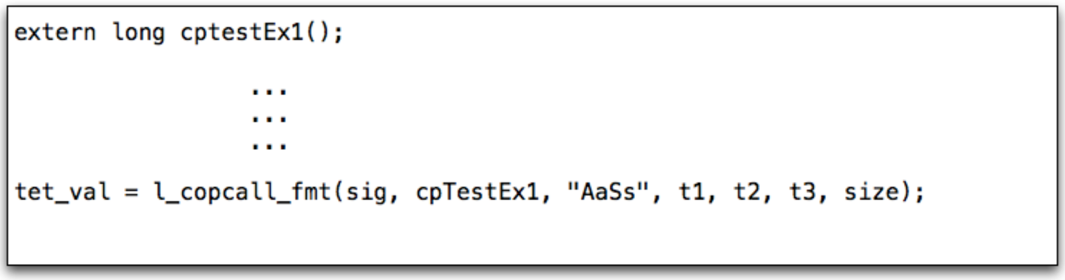
Figure 3.17 AE FPGA Floor Plan for HC-1

3.11 Host to coprocessor programming interface

In this section we will discuss the host to coprocessor programming interface, which is the way we can dispatch a function from the host to the coprocessor of the HC-1 system. Instructions that have to end up in the AEs go through the steps described below.

3.11.1 Host code

The host program is typically written in C/C++ or in Fortran [16]. If we want to make use of a custom instruction, we have to dispatch a function to the coprocessor. By using the copcall functions (provided by Convey), we dispatch our function with its necessary arguments to the coprocessor. An example of the host code dispatching a function is presented in figure 3.18 below. In this example cptestEx1 is our function that should run on the coprocessor and the sig argument indicates which personality we want to use. L_copcall_fmt is Convey's copcall function, which dispatch our function together with its arguments to the coprocessor.



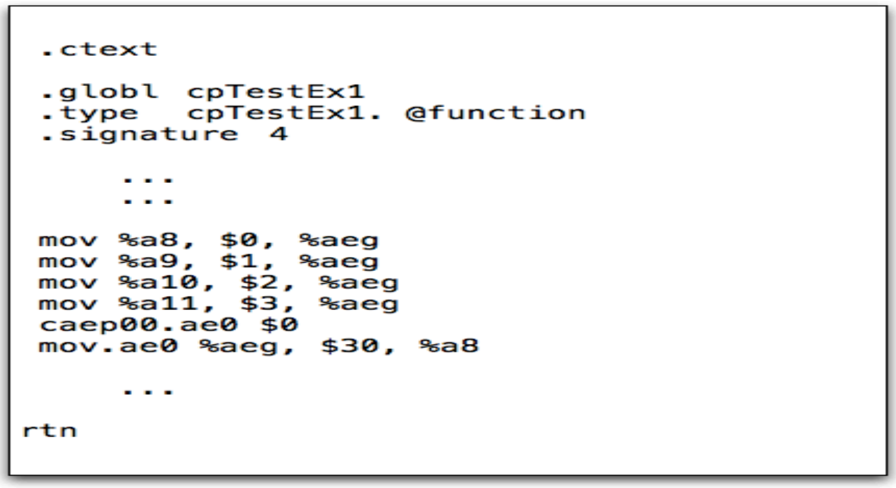
```
extern long cptestEx1();

                ...
                ...
                ...

tet_val = l_copcall_fmt(sig, cpTestEx1, "AaSs", t1, t2, t3, size);
```

Figure 3.18 Example of the host code dispatching a function to the coprocessor

3.11.2 Coprocessor code



```
  .ctext

  .globl  cpTestEx1
  .type   cpTestEx1. @function
  .signature  4

       ...
       ...

  mov %a8, $0, %aeg
  mov %a9, $1, %aeg
  mov %a10, $2, %aeg
  mov %a11, $3, %aeg
  caep00.ae0 $0
  mov.ae0 %aeg, $30, %a8

       ...

 rtn
```

Figure 3.19 Call function, which consists of instructions defined by Convey's scalar instruction set and runs on the coprocessor

The FPGA call function is executed on the coprocessor, and consists out of instructions defined by Conveys scalar instruction set. The four move instructions in the example presented in figure 3.19 above move the t1, t2 and t3 and size arguments from their A coprocessor registers to the AEG registers on the AE. The caep00 instruction calls the AE to execute custom instruction 00. Finally the last move instruction returns the value from the first AE to the register of the coprocessor's instruction set.

The coprocessor's instruction set has two types of scalar registers. The A registers which are a set of general purpose registers intended to be used to manipulate addresses for S and AE register loads and stores. Additionally, the A registers are used to handle loop counts and calculating vector length, vector stride, vector partition length and vector partition stride. The S registers are a set of general purpose registers intended to be used to manipulate scalar data [17]. The first value that is loaded into an A register is loaded into A8, the second into A9, etc. Similarly, the first value loaded into an S register is loaded into S1, the second into S2, etc.

3.11.3 AEs code

The AE receives the caep00 instruction to be executed. When the AE code finishes, it can return data to the coprocessor routine, or write a flag to memory.

4. Introduction

This chapter describes the necessary steps for the setup and the familiarization with the CONVEY tools. Also, it describes in details the hardware and software components of the first and second generation of NCBI-BLASTn personality architectures. Final, there is a brief presentation of previous works of BLAST on FPGAs.

4.1 Setup and familiarization

The setup and familiarization processes consist of two phases. We installed and experimented with both cross-development software tools provided by CONVEY and the HC-1 server itself.

First, we downloaded from CONVEY the cross-development tools and installed them to our local server. As mentioned in chapter 3, PDK development requires some extra tools, like the Xilinx ISE Design Suite 12 for synthesis, place and route of FPGAs and Mentor ModelSim HDL simulator for Verilog and VHDL simulation, as well. Second, an HC-1 server was supplied with the pre-installed native CONVEY's software tools Also, the Xilinx ISE Design Suite 12 tool was used for HDL synthesis. Finally, the Synopsys VCS tool was used for hardware simulation after the setup of it's license server.

Acquaintance with the CONVEY tools started with working basically with CONVEY's provided examples. Our main interest was on PDK tool and its examples. First, we spent some time to deal with the provided examples dedicated to HC-1's programming tools, which use single and double precision personalities as well as financial analytics personality. Valuable experience was gained on CONVEY's compilers, debugger and simulator (mainly used due to not having license for the above personalities).

The training effort on developing custom personalities for HC-1 was based on PDK's sample personality. A detailed and deep analysis of this example personality in combination with a variety of small custom personalities, which were produced by changes made on the sample personality, led to deep understanding of CONVEY's step-by-step instructions for developing a custom personality for its Coprocessor.

4.1.1 Sample personality

CONVEY has developed a sample custom personality, as part of the PDK package, that can be used as a reference design. The sample personality was designed to be very simple while using all of the required hardware interfaces inside the FPGA except of the AE-to-AE interface.

CONVEY's sample personality implements a memory-to-memory instruction that adds values stored in one block of memory to the values stored in another block of memory. The array of results is stored back in memory. It consists of 16 copies of a functional block, which adds a portion of the memory storing the operands. The functional blocks are directly connected to the memory controllers so that each block only performs operations on values in its attached memory. A detailed block diagram is presented in figure 4.1 below. The sample personality was used as a starting point for new PDK personalities.

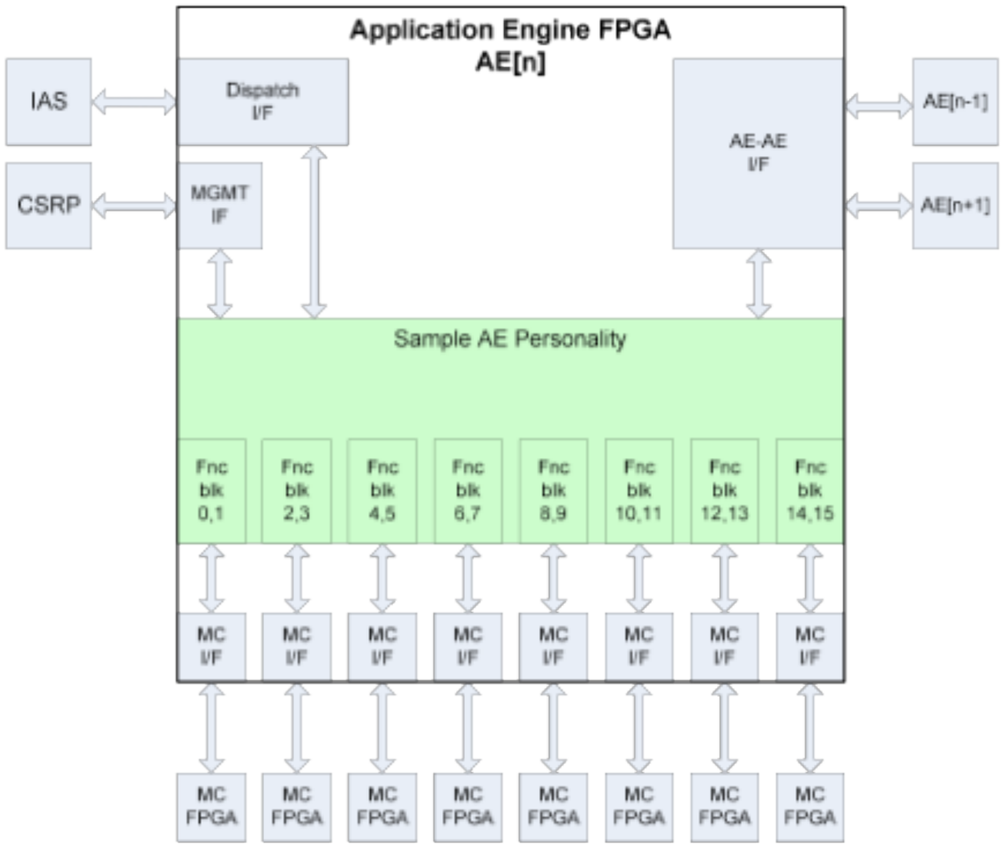


Figure 4.1: Block diagram of Sample personality provided by convey

4.1.2 PDK tools familiarization

Some new variations of the sample personality were developed for better understanding of the sample personality's functionality and pdk's interface connections. We made small changes experimenting with both the simulator's golden model and the Verilog designs.

The first variation is very small and includes only the functionality of the sample personality. We changed the sample personality from a memory-to-memory adder, to a memory-to-memory substractor. The changes took place to the host processor's code, to the simulator's model and to the hardware design. During the development of this variation we had the chance to learn how to create a new personality and get a deeper understanding of the sample personality. Our second try was to create a custom application connected only to the dispatch interface. We implemented a basic scalar adder in Verilog and connected it with the dispatch interface. That gave us a better view of the way the custom application interacts with the scalar processor of the coprocessor, how to develop software model for the simulator and how to dump and visualize hardware signals simulation. Next variation helped us find how to use pre existing cores generated by Xilinx's Core Generator. We generated a Block RAM with Xilinx's tool, which we integrated, with our custom designed fine state machine (FSM) creating a simple personality, which takes an address and returns its content.

In general, we developed even more variations of the sample personality, all of them on the cross-development tools and using the simulators mentioned above.

4.2 Previous work on BLAST

Biological sequence searching using FPGAs has been an active area of research for over a decade. A lot of research has been done in implementing BLAST.

BLASTn's bottleneck resides in the first stage where the hits are located. There are two different classes of BLASTn works. In the first one belongs the works, which are mainly focused in the bottleneck, and in the second one belongs the works, which implements the entire algorithm on the FPGA. A brief description of some of this works from both classes follows.

The first one is the RC – BLAST project [18], which belongs to the first class of solutions, but its overall performance was reported to be poor, even worse than the corresponding software implementation and no further efforts where given for this project. Despite that, the RC – BLAST project remains important as it is the first full BLAST implementation in reconfigurable technology.

Next one is the TreeBLAST project witch was presented in 2005 by the CAAD Lab at Boston University. This project belongs to the second class of solutions. It is a BLAST algorithm implementation for small queries of up to 800 nucleotides [19] and it was extended and implemented later [20], providing speedup vs. the software implementation. An extension of TreeBLAST presented in 2009 [21] was based on the database prefiltering.

The parallel Mercury BLAST [22], [23] [24] is one more implementation of BLASTn introduced by Washington University in St. Louis, which offers a good speedup vs. software execution on a general purpose computer.

One more completed system work is TUC-BLAST [25], which implements BLAST algorithm for all its versions and for any size of database and query. This system has been fully designed and partially implemented using FPGAs and it consists of software and hardware parts. This implementation achieved speedup of up to thousands of times vs. general purpose computers.

Multi-seed/ Multi-channel BLAST [26] [27] is the most recent effort from Chinese National University of Defense Technology which also reports very interesting results for a generic architecture for BLAST algorithm.

Except of the above academic approaches there are and some commercial companies, like Time Logic [28] and SGI/Mitrionics, which offer the Decypher machine and RASC Appliance respectively. They offer high performance over present multi core processors but they provide little detail about their designs. Recently, the IBM Cell broadband Engine [29] has also been used to speedup BLASTn.

Final, the related work on BLAST CONVEYs BLAST personality presented in [chapter 3 ready personalities] can not be omitted.

4.3 System design and major decisions

This section describes the implementation of the NCBI-BLASTn personality. The HC-1 hybrid supercomputer was used for the BLAST implementation. We implemented a hardware design to run on the coprocessor, and a software program controlling the coprocessor and managing IO. Some major decisions are listed below, which were taken before and during the implementation of the personality.

4.3.1 Development process

We proceeded to the implementation in phases, which led to the generation of two different NCBI BLASTn personalities. This approach helped us discover and make clear implementation issues, coming up during the development process, individually, and as fast as possible.

4.3.2 HW-SW co-design

The main target was to develop the personality that would substitute the most time consuming function of NCBI BLASTn software implementation.

After a large number and many extensive, profiling tests the three different functions s_BlastSmallNaScanSubject_8_4, SMALL_NA_ACCESS_HITS, s_BlastSmallNaRetrieveHits were chosen for hardware implementation. These functions implement the seeding step of the algorithm, as mentioned in Chapter 2. We decided to implement in hardware the last two and part of the first one, and in software the communication between our hardware design and NCBI's software.

4.3.3 Implementation's fidelity

The most important target of the implementation was to have identical results vs. the NCBI's official software.

Although, BLAST is a heuristic algorithm and a hundred percent compatibility with NCBI's blast output is unnecessary, it is difficult to convince biologists of the same. A typical user would have no idea whether the differences are statistically significant. That's why we decided not to compromise fidelity of our design as other implementations do. Due to this decision, we left aside the parallelization factor and also the speed up that it produces during the first phases of our design process.

4.3.4 Data structures Design

Another issue that came up from the start due to the integration mentioned above, was the design of the data structures, which affect the efficiency of parallelization and overall implementation. For this issue, both the software and the hardware of the implemented system have their own data structures and also some shared ones. The most important data structure, which is used by our implementation consist of a query-index table named backbone and an overflow table storing the positions of the words of the query, a table that stores the database subjects and an index table storing the resulting ungapped alignments between the query and each of the database subjects. The location of each data structure in memory was carefully selected, depending on data size and how often each structure is accessed during execution. In particular, frequently accessed structures are stored in the possibly fastest memories that could accommodate their size.

4.3.5 Testing data sets

For testing and evaluation reasons, small testing datasets were created, which are much smaller in size but accurate enough to help us accelerate development process and evaluate the functionality and output fidelity of our implementation. These datasets include a variety of input databases with different number of subject sequences of different lengths, which are formatted with the help of NCBI’s formatdb program. Also, two reference query sequences were used for building backbone and overflow data structures mentioned above. The first query sequence was small but really helpful during functional evaluation and the other one larger and more representative of the average query lengths.

4.4 BLASTn personality architecture: 1st generation

This section presents the first generation architecture of our personality in details. It was developed with all decisions described in previews section and it was also our first attempt to develop a hardware extension of a well known and established software, in the community of biologists, on a very high tech server.

The implementation is a prototype, which maps part of the functionality of the seeding step. It takes as input from the NCBI-BLASTn software a sixteen-bit quantity representing eight residues of the subject sequence, and counts the number of appearances of this quantity in the query, and more specifically in the pre-generated memories where the w-mers are kept. This number is what it returns to the software so that the software can resume the execution. An abstract block diagram of the architecture is presented in the figure 4.2 below.

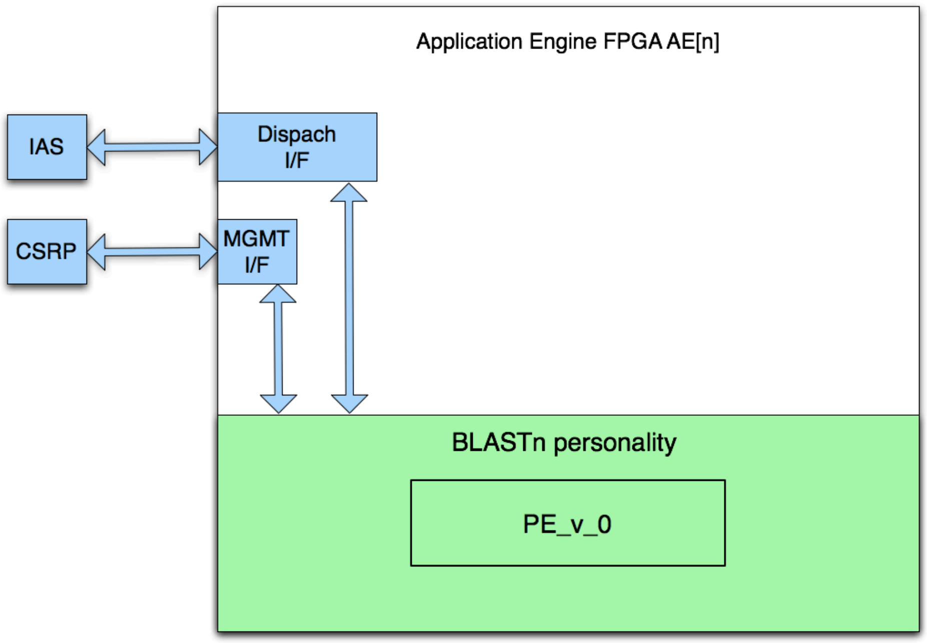


Figure 4.2: Block diagram of 1st generation BLASTn personality

As shown in the figure above the implementation uses only of dispatch and the processor management interfaces. Figure 4.3 gives a more detailed view of the design and how it uses these interfaces.

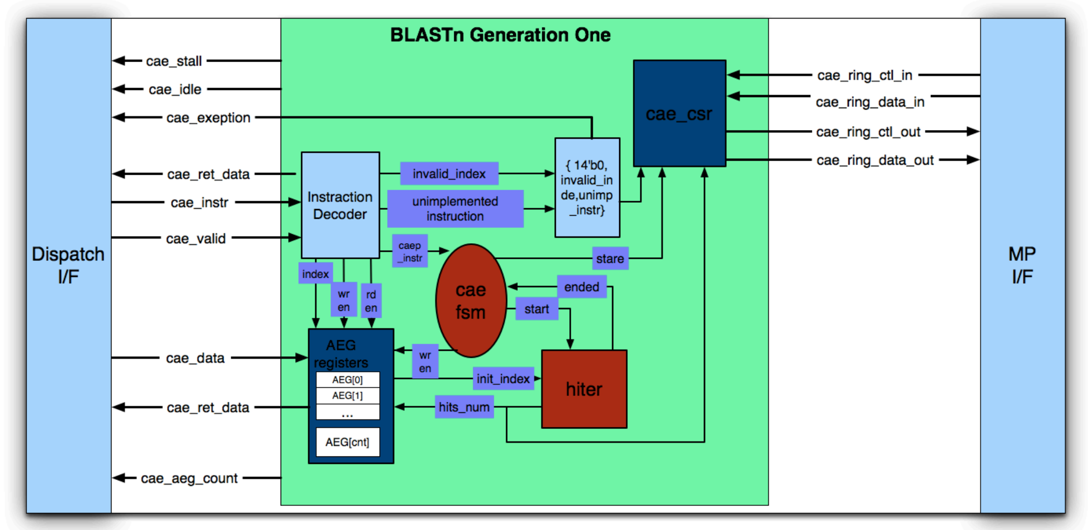


Figure 4.3: Detailed block diagram of 1st generation BLASTn personality showing its connection with the PDK's interfaces

4.4.1 AEG registers definition

The personality consists of the logic needed to decode the instructions coming from the dispatch interface. These instructions either move data from the application engine to the dispatch interface and vice versa or implement a custom instruction. This logic also generates an exception if an instruction is not implemented in the design.

AEG registers are used for storing the data, which is moved from or to the application engine. The implemented infrastructure uses only two AEG registers, which are defined at the table 3.1 below.

AEG index	Register Name	Description
0	Init_index	Hiter initial index
30	Hits_num	Hiter output

Table 3.1: AEG registers

Init_Index register

Init_Index register stores the sixteen less significant bits of the subject sequence, which is checkedfor existence in the look up tables.

Hits_num register

Hits_num register stores the output of the hitter module when it finishes. The value of this register is returned to the host processor.

If an invalid index of an AEG register is given to the design it will produce an exception. The processing element of our design is the hitter module.

4.4.2 Hitter module version one

The hitter module implements most of the work. Custom instruction zero enables the hitter module to start processing. A detailed figure 4.4 of hitter module architecture follows.

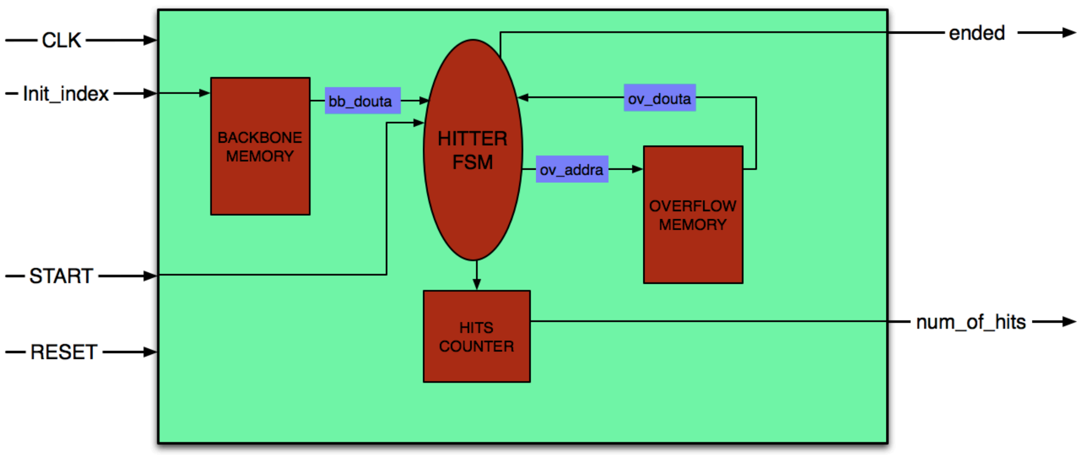


Figure 4.4: Hitter module generation one block diagram

The hitter module consists of four parts. These are two BRAMs of size 65536 X 16 each, which stores the backbone and overflow memory respectively, a Fine State Machine (FSM) that is responsible for the control of the module and a counter for counting the number of hits.

The two memories are the data structures accessed most often, almost during every clock cycle, that's why we decided despite their size to place them in BRAMS inside the Application Engine FPGAs and initialize them with two pre-generated "COE" files in which we saved backbone and overflow tables generated by NCBI-BLASTn software execution for a specific query. As this design is a lab prototype we chose to stick with those two predefined memories, so we properly chose to use a query, which would be representative of most nucleotide sequences regarding the sequence length.

The FSM control unit defines the correct signals for all the functional units of the module. It has four states, which compose the whole functionality of the module; transactions between stages are shown in figure 4.5 below.

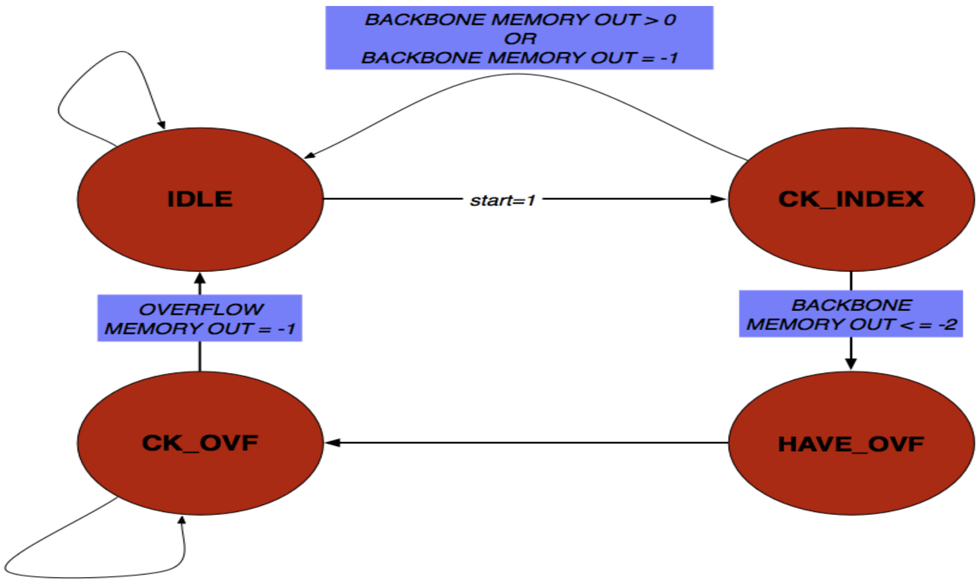


Figure 4.5: Hitter module control FSM stages transactions

4.4.3 Hitter module functionality

Hitter's module functionality in the first generation architecture is to count the number of seeds for every sixteen-bit subject sequence quantity (init_index) in the query. At the time init_index is available and START bit is set to one, the hitter uses the init_index as a memory read address for the backbone memory. Its output is used by the FSM to check if the index has zero, one or more appearances in the query. When the backbone memory returns a minus one value, this shows that the searching index has zero appearances. In case, the memory's returned value is an integer greater than or equal to zero, the index has only one appearance in that position and the FSM increases the counter by one end sets ended bit to one, which indicates the end of seeds searching for this particular index. On any other case which means it is a negative integer less than or equal minus two, the index has more than one appearances. In such a case the FSM gets the absolute value of this negative integer and uses it as a base read address for the overflow memory. At that point, a loop depending on overflow's output starts. Initially, the overflow's position holds the number that represent the position of the first appearance of the index in the query and overflow's next position holds the second number and so on until overflow's output takes the value minus one and the loop terminates. Each iteration of this loop takes one cycle time to complete and in each iteration the counter increments by one except the last one where ended output signal is set to one. When ended signal is set, the num_of_hits holds the number of the seeds for this index.

4.4.4 CAE CONTROL FSM

The control FSM for the custom application engine is responsible for enabling the hitter module and store its output result in the appropriate AEG register. Also, it controls the stall and the idle signals from the dispatch interface. An abstract diagram of its stages transactions is presented in figure 4.6 below.

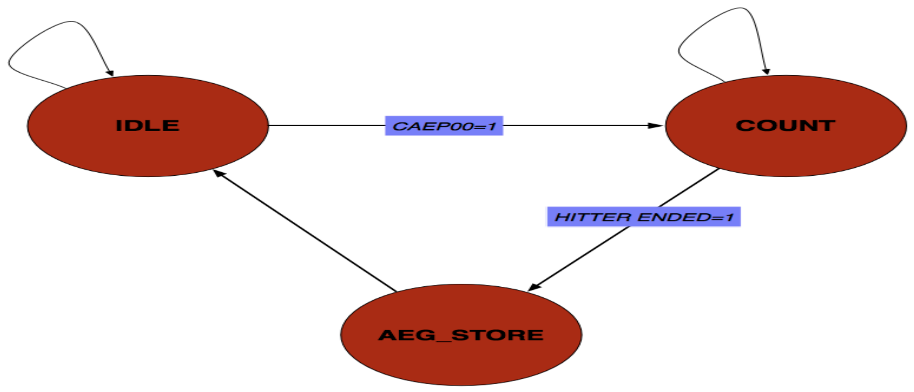


Figure 4.6: CAE control FSM stages transactions

4.4.5 Host application design

A program, which is a modified version of the original NCBI BLASTn and runs on the Intel Xeon processor, controls the coprocessor. The main difference between the original program and the one developed to use the implemented personality is mostly on the three functions that have already been described in chapter two. More specifically, for the first generation of the design the differences are only in s_BlastSmallNaScanSubject_8_4 seeding function. We changed this function by adding a portion of a code responsible for loading our personality to the coprocessor and also managed the dispatch to it by calling the assembly code with the necessary input arguments. A flow chart of the modified function is presented in figure 4.7 below. The only difference from the original flow chart presented in chapter two figure 2.16 is the addition of the copcall function, which makes a dispatch to the coprocessor and loads the assembly written external function to it with only input argument, a part of the subject sequence of eight letters length (sixteen bits). This function returns in num_of_hits variable the number of hits counted by our personality for this particular index.

The main part of the assembly external function above is loading the init_index value to AEG [0] register described in details above and makes a call to the custom instruction zero (caep00) which with its turn enables the Application Engines mapping our design. When our design ends the execution, the counting result that is in AEG [30] is returned with assembly instruction to the host processor. During the execution of the external function on the coprocessor, the host program blocks its execution waiting for the result. When the return happens the host continues its execution from the point it was blocked.

In this first generation of our implementation, the return value of the hardware part in num_of_hits variable is used only for evaluation reasons.

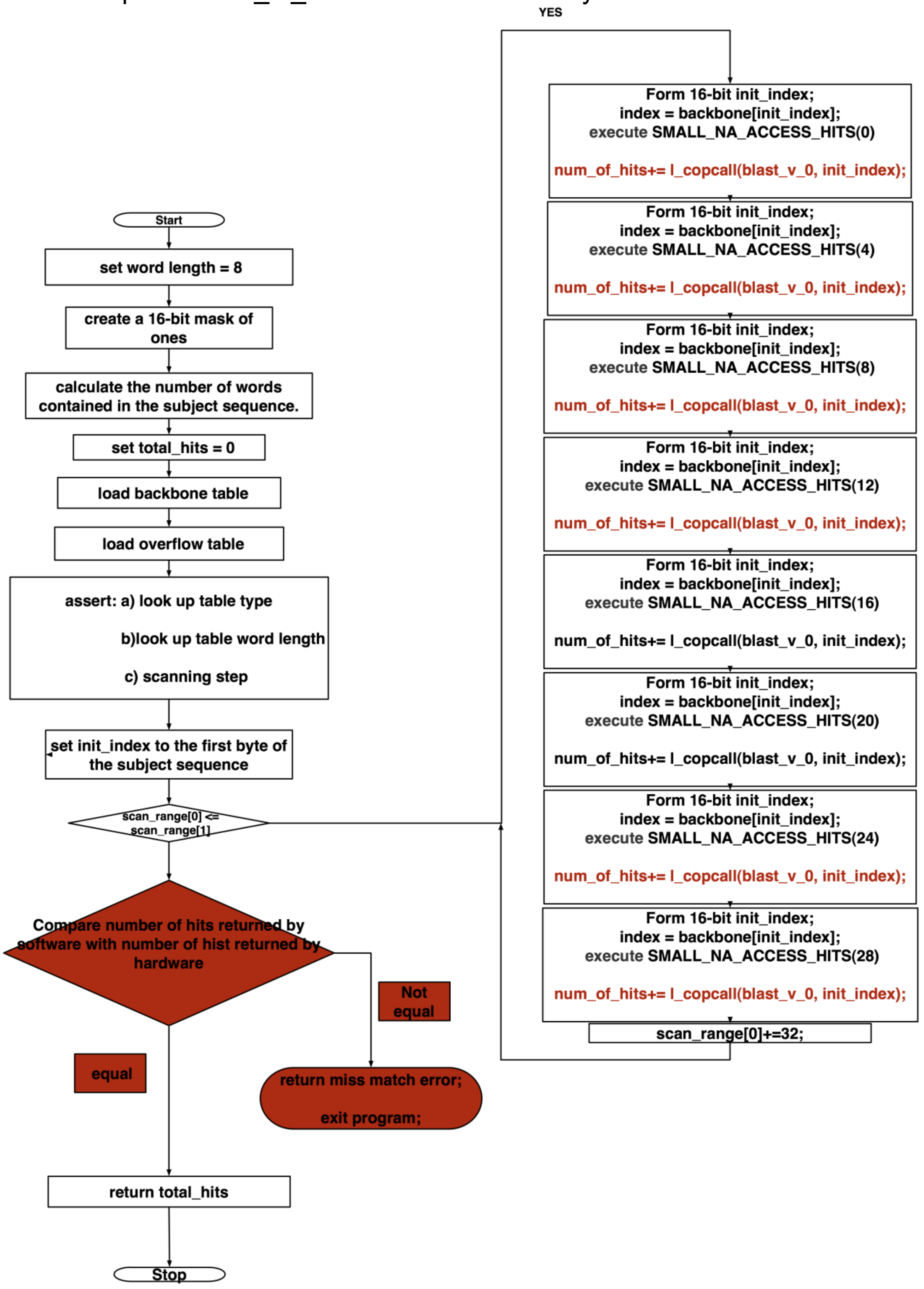

YES
Start
set word length = 8
create a 16-bit mask of ones
calculate the number of words contained in the subject sequence.
set total_hits = 0
load backbone table
load overflow table
assert: a) look up table type
b)look up table word length
c) scanning step
set init_index to the first byte of the subject sequence
scan_range[0] <= scan_range[1]
Compare number of hits returned by software with number of hist returned by hardware
Not equal
equal
return miss match error;
exit program;
return total_hits
Stop
Form 16-bit init_index;
index = backbone[init_index];
execute SMALL_NA_ACCESS_HITS(0)
num_of_hits+= l_copcall(blast_v_0, init_index);
execute SMALL_NA_ACCESS_HITS(4)
execute SMALL_NA_ACCESS_HITS(8)
execute SMALL_NA_ACCESS_HITS(12)
execute SMALL_NA_ACCESS_HITS(16)
execute SMALL_NA_ACCESS_HITS(20)
execute SMALL_NA_ACCESS_HITS(24)
execute SMALL_NA_ACCESS_HITS(28)
scan_range[0]+=32;

Figure 4.7: Block diagram of 2nd generation BLASTn personality

First generation BlASTn personality was a lab prototype that we mostly used for getting the experience of developing a hardware and software co-design implementation based on very complicated software like the NCBI-BLAST. It helped us get more experience on the tools and especially on debugging large and complex designs. It also helped us on getting answers and solutions on questions and teething problems, mostly on tools usability, coming up during its design.

Although the first generation personality design is a conservative approach to the real problem, it gave us a satisfactory base to work on, as long as it has been fully functional and accurate. Further work and evolution development, resulted in the second generation described below.

4.5 BLASTn personality architecture: 2nd generation

This section describes the second generation architecture of NCBI Blast personality, which is a newer version of the first one. The main aim concerning this generation was to extend first generation's functionality one step further. We used the memory controllers and their interfaces so that the Application Engine stores its query end its subject sequence offset index for every hit it finds. Due to the fact that every memory controller is connected to the one eighth of the memory and we want to write on different places on the whole memory we decided to use the memory crossbar.

The memory crossbar is a Convey provided module instantiated between our design and the memory controllers interfaces on the Application Engine and routes our memory requests to the right interface which will send it to the right memory controller.

An abstract block diagram of the architecture is presented in figure 4.8 below where a clear view of Application Engine's modules placement is presented.

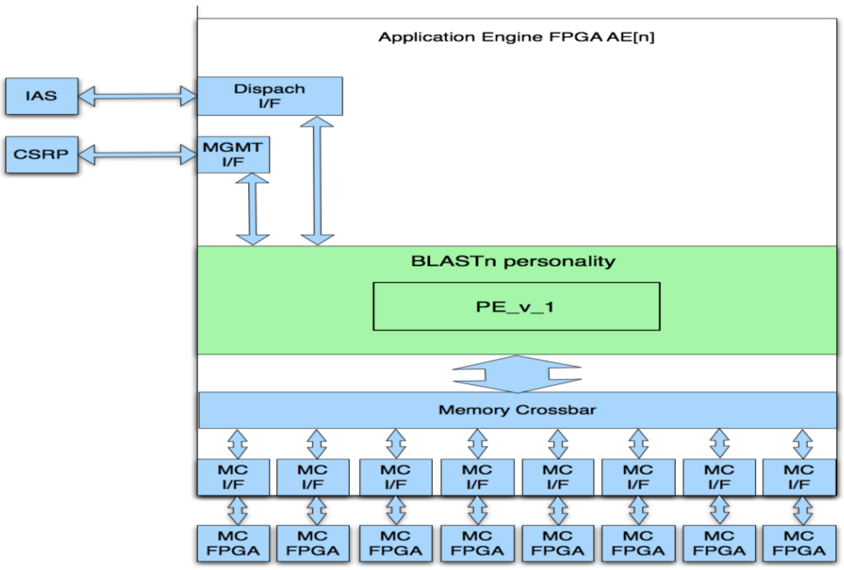


Figure 4.8: Block diagram of 2nd generation BLASTn personality

A more detailed figure 4.9 showing the interconnection between PDK's interfaces and BLASTn second generation personality follows. This diagram also shows the inner parts of our second-generation personality.

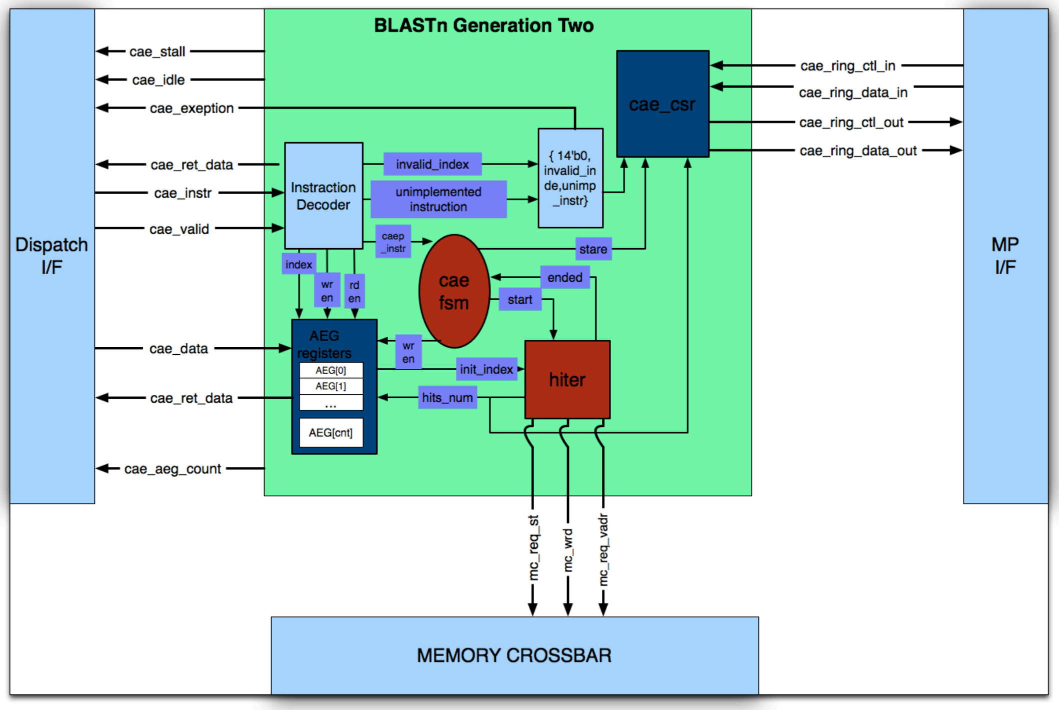


Figure 4.9: Detailed block diagram of 2nd generation BLASTn personality showing its connection with the PDK's interfaces

4.5.1 AEG registers definition

The AEG registers that were used for this generation are defined at the table 4.2 below.

AEG index	Register Name	Description
0	Init_index	Hiter initial index
1	mem_base	Base write memory address for storing hits
2	s_off	offset in subject sequence
30	Hits_num	Num of hits (Hitter output)

Table 4.2: AEG registers definition

Init_Index register and Hits_num register

These two registers are exactly the same with those used in generation one.

Mem_base register

Mem_base register is used to store the starting address from where the Application Engine will start storing results.

S_off register

This register is used to hold the offset index in the subject sequence. It is used to find hit forms on an offset pair index by concatenating its value with the value that the hitter retrieves from its memories.

Exceptions for this generation are the same with the previous.

4.5.2 Hitter module changes

The hitter module architecture went under few changes, which were enough to add the extra functionality we were looking for. The basic modules left untouched but the most variations have been made mostly on the FSM controlling the hitter module and adding more functionality in it without changing stages and their transactions. An additional offset pair write address generator was implemented as Figure 4.10 shows.

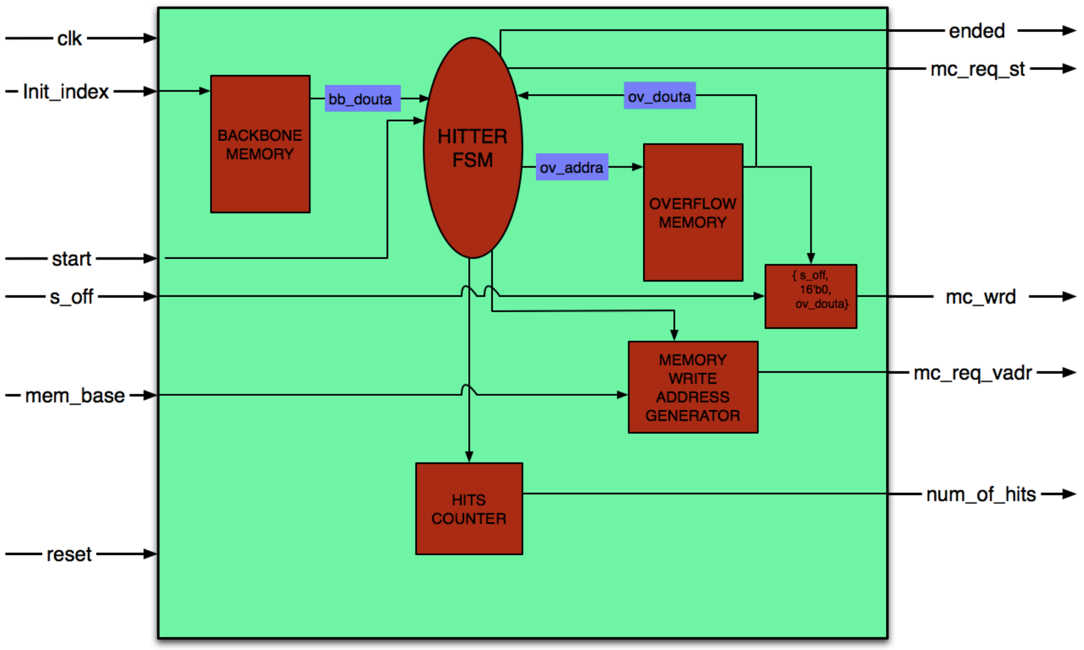


Figure 4.10: Hitter module generation one block diagram

4.5.3 Hitter functionality

Hitter module in this second generation architecture keeps all the functionality of hitter module used in previous generation as it is an evolution of it, but there is some additional functionality, as this hitter is capable of storing query and subject sequence offset indexes, for every hit it finds, directly to memory.

For every sixteen bit quantity passed to hitter we also pass its offset index in the subject sequence which is thirty two bits long. The hitter module uses this extra input; in concatenation with the unsigned extended query offset index retrieved from the hitters memories (memories outputs are sixteen bits long) for every hit it is found to form a sixty four bit quantity representing an offset pair which will be used from the software responsible for the extension step of the program. Each offset pair is sent together with the memory address generated from the memory address generator and the necessary write control signals to the memory crossbar that was mentioned above.

When the hitter retrieves all hits, it raises the ended signal and num_of_hits holds the number of hits found for this quantity.

4.5.4 CAE control FSM

This module in second generation architecture have been left unattached as it was in the first one, figure 4.5.

4.5.5 Host application changes

For this generation the host application has more changes. First, the assembly external function has changed and became more complicated, in this version as it loads not only AEG [0] but also AEG [1] and AEG [2]. The rest of the function has no changes. Second, the NCBI BLAST source code has come under changes unlike the first generations. These changes are mostly on removing part of the code which now is replaced by the hardware. In details, this replaced code matches to the two functions SMALL_NA_ACCESS_HITS and s_BlastSmallNaRetrieveHits which are called from the program for every sixteen bit quantity. The S_BlastSmallNaScanSubject_8_4 function has also changed, now it's responsible to form the sixteen bit index and make a call to the hardware providing the necessary arguments.

4.6 BLASTn Implementations differences

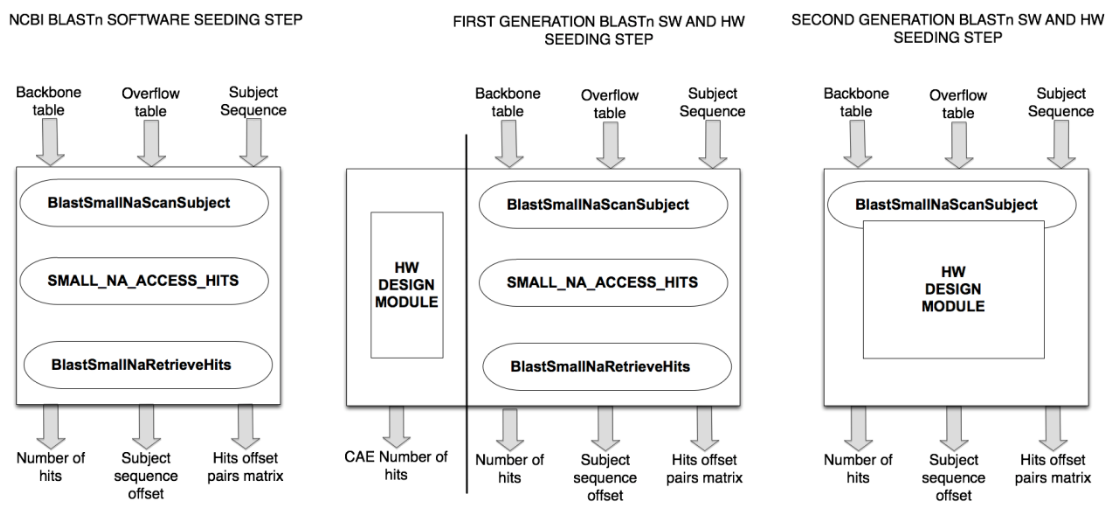


Figure 4.11: BLASTn seeding step differences between NCBI's software and first and second generation personality implementations

As it is already mentioned in session 4.4, in the first generation architecture our custom application engine (CAE) which is the hardware design module placed in the FPGAs, is running in parallel with the host program's seeding functions without effecting their functionality, and outputs the number of hits, only for comparison reasons, with the one produced by these functions. Unlike the first generation, in the second one the CAE is a functional part of the program and it replaces two of these functions and part of the third one. Differences between NCBI's BLASTn program and the two generations software and hardware co-designs we have developed according to the seeding step is shown in figure 4.11 above.

The second generation’s personality CAE output plays a vital role in the alignment output of BLASTn that’s why we have spent a large amount of time on testing and evaluating since we have been aiming at an one hundred percentage of matching between our personality alignment and the alignment taken from the NCBI’s software.

5. Performance and validation

According to chapter two, the BLAST algorithm consists of three basic steps: seeding, extension and evaluation. The profiling of the NCBI BLASTn program showed that the hotspot of the algorithm is the seeding process, which consumes up to fifty percent of total execution time.

As chapter four describes, the most time consuming functions of the seeding process were implemented and mapped in reconfigurable hardware. This chapter describes the performance of the second generation NCBI BLASTn personality implemented. First generation's personality was used most for familiarization and evaluation reasons and that's why its performance was not taken under evaluation. This section compares the software stand-alone NCBI BLASTn program vs. the software hardware co-design implementation.

For both implementations the performance tests were based in one query, which was properly chosen so that its length to be representative of the average nucleotide query length, and on four different databases of various number of sequences and lengths. Also, the performance results are based on "wall time" as both implementations are fully functional. Tables 5.1 and 5.2 show the size of the testing query and the four databases.

QUERY SEQUENCE LENGTH(letters)	782

Table 5.1: Testing query sequence length in letters.

DATABASE NAME	No OF SEQUENCES	No OF LETTERS
Test_db	6	4309
Testdb_large	1	3.756.989
Env_nt	18.486.260	7.655.037.578
nt	14.245.355	36.716.926.086

Table 5.2: Testing databases No of sequences and No of letters.

5.1 System validation

As already mentioned many times above, the main goal of the software-hardware implementation was the validation of its output vs. the software implementation's results. Many test have shown that this goal was achieved a hundred percent, as the results of both implementations are identical.

5.2 Resources utilization

This section analyzes the recourses of the HC-1 system and the utilization of our second generation BLASTn implementation. Although the available hardware resources are four AEs (FPGAs), our design maps only to one of them. Table 5.3 shows the utilization of the FPGA for one Hitter Processing Element (HPE) and table 5.4 shows utilization of the whole personality, using one processing element.

Hitter Processing Element recourses utilization (one Virtex5)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	139	51840	0%
Number of Block RAM/FIFO (36 Kb)	45	288	15%
Number of DSPs	0	192	0%

Table 5.3: HPE resources utilization.

BLASTn 2nd generation arcitecture recourses utilization (one Virtex5)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	15364	51840	29%
Number of Block RAM/FIFO (36 Kb)	87	288	30%
Number of DSPs	0	192	0%

Table 5.4: 2nd generation BLASTn personality resources utilization.

Table 5.3 makes clear that the critical resource of the HPE is the Block RAMs. The above numbers reveal that the extra logic of the whole system including PDK's interfaces consumes 42 Block RAMs leaving 246 for adding more HPEs working on parallel.

HC-1 system can combine 5 parallel working HPEs on each of its AE, summing up to 20 HPEs total.

5.3 System performance

This section describes the performance analysis of our BLASTn system implementation and where it stands in comparison with the performance of the software implementation provided by NCBI.

It is necessary to point out that our implementation is not a design focused on performance of BLASTn but a design of fully functionality and accuracy. As the sw-hw implementation and the NCBI Blast software are complete systems, the "wall time" measurement was used for performance comparison.

The software execution was tested in the HC-1's host processor, which is a 2,13 GHz dual core Intel processor.

Table 5.5 presents the "wall time" for the seeding step that consists of three different software functions, which were mapped in the reconfigurablke logic. Table 5.6 presents the "wall time" for the entire systems.

Functions: s_BlastSmallNaScanSubject_8_4, SMALL_NA_ACCESS_HITS, s_BlastSmallNaRetrieveHits (seeding step)				
Database	**SW Performance (sec)**	**SW-HW Performance (sec)**	**Speedup (SW vs. SW-HW)**	**No of coprocessor calls**
test_db	0.01	0.9	90	1069
testdb_large	0.01	10.86	1086	939246
env_nt	10.7	10577.33	988	1888341885
nt	35.46	104969.75	2960	569752874

Table 5.5: Wall time performance of seeding step both in NCBI's BLASTn software implementation and 2nd generation BLASTn

BLASTn implementations (SW by NCBI SW-HW by us)			
Database	**SW Performance (sec)**	**SW-HW Performance (sec)**	**Speedup (SW vs. SW-HW)**
test_db	0.01	0.9	90
testdb_large	0.019	11.53	607
env_nt	16.089	21942	1371
nt	251	106055	423

Table 5.6: Wall time performance of both in NCBI's BLASTn software implementation and 2nd generation BLASTn

5.3.1 Timing analysis

The comparison of the perfromance results, which are presented above, shows that our implementation is much slower than the software one. This is due to two very significant reasons.

Parallelization:

As already mentioned, our implementation maps only 1 HPE. The reason why only one is used is mainly validation reasons, as we wanted to create a basic design, which will be easily checked as far as its accuracy concerns. HPE's clock frequency is 150 MHz a number due to the Convey's system specifications.

Coprocessor calls:

For both implementation's execution time varies a lot depending on the database and more specifically on the number of letters it has, as for every 8 letters (sixteen bit quantity) a call to the seeding functions has to be made. Our implementation makes a call to the coprocessor and for every call it has to pass the necessary arguments for the hardware implementation initialization through assembly instructions to the coprocessor, which are executed to the coprocessors scalar processor, which works on really, lower clock frequency than the host. All this coprocessor calls adds a large time overhead in our implementation.

In next chapter there are some proposals how to overcome this two bottlenecks of our design in a future work.

6. Conclusion and future work

This chapter concludes this thesis and it, also, describes some proposals for future work based on what have already been done.

6.1 Conclusion

This thesis analyzed the software implementation of the Blast algorithm provided by NCBI and deep knowledge of the BLAST algorithm was gained. Although we spend time on BLAST algorithm's ideas most of our work was on a specific version of it, the nucleotide BLAST (BLASTn). We also developed two hardware software co-design implementations based on it and mapped them in a state of the art heterogeneous computing system. These two implementations are different steps of continuous development and evolution resulting in a functional prototype of BLASTn application running on our system. We achieved our goals concerning training, experience gaining with this complicated high performance system and fidelity of our implementation, there are some future work development proposals focused mostly performance.

6.2 Future work

Generation two of the BLASTn application, which derived from the first one, has left some positive signs with its fidelity but it lucks performance. The system's potentials, especially the memory system potentials, and also the parallelization hidden in the algorithm seem encouraging and raise some aspects for future work. Some proposals are:

- Migrating the functionality of generating input data and passing them to the AEG registers of the CAE through execution of scalar instructions and system function calls to the CAE itself, so that a hardware module will replace all three software functions and take full advantage of the system's memory bandwidth.

- Adding more than one hitter module, which will take fully the advantage of the algorithm's parallelization and will make use of the great variety of system recourses.

- More extra functionality concerning the extension step can be added to the hardware. In this case, more steps will be executed in parallel.

REFERENCES

[1] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

[2] *Prasanna Sundararajan, "High Performance Computing Using FPGAs", Xilinx* White Paper 375 (v1.0) September 10, 2010.

[3] Altschul,S.F. *et al,* "Basic local alignment search tool", *J. Mol. Biol.*, 215, 403–410, 1990

[4] Peters,R. and Sikorski,R, "BLAST off! *Science"*, 278, 510–502,1997

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

[6] Smith,T. and Waterman,M, "Identification of common molecular subsequences", *J. Mol. Biol.*, 137, 195–197.

[7] Karlin,S. and Altschul,S.F, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes", *Proc. Natl Acad. Sci. USA*, 87, 2264–2268.

[8] M. Dayhoff, R. Schwartz, and B. Orcutt, "A model of evolutionary change in proteins," In: Dayhoff,M,O, (ed), Atlas of Protein Sequence and Structure National Biomedical Research Foundation, Washington, DC, pp, 345–352, 1978.

[9] S. Henikoff, J. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks", PNAS 89: 10915–10919, 1992.

[10] Cynthia Gibas, Per Jambeck, "Developing Bioinformatics Computer Skills"

[11] Jason Papadopoulos, "The Developer's Guide to BLAST "

[12] Convey, "Convey Personality Development Kit Reference Manual", Version 4.2,June 2011

[13] Jason D. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1", 1-1-2010

[14] Karl Savio Pimenta Pereira, “Characterization of FPGA-based High Performance Computers”

[15] Convey, ”Convey Spat Users Guide”, Version 1.0, June 2009 .

[16] Convey, “Convey Programmers Guide”, Version 1.8, November 2010.

[17] Convey, “Convey Reference Manual”, Version 2.00, September 2009.

[18] K. Muriki, K. Underwood, and R. Sass, “RC-BLAST: Towards an open source hardware implementation,” In Proceedings of the International Workshop on High Performance Computational Biology (2005).

[19] M. Herbordt, J. Model, Y. Gu, B. Sukhwani, T. VanCourt, "Single Pass, BLAST-Like, Approximate String Matching on FPGAs"14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), pp, 217-226, 2006.

[20] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, “Single pass streaming BLAST on FPGAs”, Parallel Computing, vol. 33, issue 10-11 (Nov, 2007), pp 741-756, 2007.

[21] P. Afratis, E. Sotiriades, G. Chrysos, S. Fytraki, and D. Pnevmatikatos, “A rate-based prefiltering approach to BLAST acceleration,” in Proc,IEEE Conference on Field Programmable Logic and Applications, 2008.

[22] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster, "Biosequence Similarity Search on the Mercury System," In Proc. of the IEEE 15th Int'l Conf, on Application-Specific Systems, Architectures and Processors, September 2004, pp, 365-375.

[23] 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

[24] A. Buhler et al. "Mercury blastn: faster dna sequence comparison using a streaming hardware architecture", RSSI, 2007.

[25] Euripides Sotiriades, “*Reconfigurable Architecture Structures for the BLAST DNA Sequencing Algorithm”, phd thesis.*

[26] F. Xia, Y. Dou and J. Xu, “Families of FPGA-Based Accelerators for BLAST Algorithm with Multi-seeds Detection and Parallel Extension”, Bioinformatics Research and Development, Second International Conference, BIRD 2008, pp, 43-57, Vienna, Austria, July 7-9, 2008.

[27] F. Xia, Y. Dou, J. Xu, "FPGA-Based Accelerators for BLAST Families with Multi-Seeds Detection and Parallel Extension," The 2nd International Conference on Bioinformatics and Biomedical Engineering, 2008, ICBBE 2008,, pp,58-62, 16-18 May 2008.

[28] www.timelogic.com. Time logic biocomputing solutions.

[29] A. Wirawan, K. C. Keong, and B. Schmidt. Parallel dna sequence alignment on the cell broadband engine. Lecture Notes in Computer Science, pages 1249–1256, 2007.