Boston University

College of Engineering

Dissertation

FPGA Acceleration of Sequence Analysis Tools in Bioinformatics

by

Atabak Mahram

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2013

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Ayse Coskun , PhD
Professor of Electrical and Computer Engineering

Third Reader

Douglas Densmore , PhD
Professor of Electrical and Computer Engineering

Forth Reader

Allyn Hubbard , PhD
Professor of Electrical and Computer Engineering

Contents

FPGA Acceleration of Sequence Analysis Tools in Bioinformatics

ATABAK MAHRAM

Boston University, College of Engineering, 2013

Major Professor: Martin C. Herbordt, PhD,

Professor of Electrical and Computer Engineering

Abstract

With advances in biotechnology and computing power, biological data are being produced at an exceptional rate. The purpose of this study is to analyze the application of FPGAs to accelerate high impact production biosequence analysis tools. Compared with other alternatives, FPGAs offer huge compute power, lower power consumption, and reasonable flexibility.

BLAST has become the de facto standard in bioinformatic approximate string matching and so its acceleration is of fundamental importance. It is a complex highly-optimized system, consisting of tens of thousands of lines of code and a large number of heuristics. Our idea is to emulate the main phases of its algorithm on FPGA. Utilizing our FPGA engine, we quickly reduce the size of the database to a small fraction, and then use the original code to process the query. Using a standard FPGA-based system, we achieved 12x speedup over a highly optimized multithread reference code.

Multiple Sequence Alignment (MSA)--the extension of pairwise Sequence Alignment to multiple Sequences--is critical to solve many biological problems. Previous attempts to accelerate Clustal-W, the most commonly used MSA code, have directly mapped a portion of the code to the FPGA. We use a new approach: we apply prefiltering of the kind commonly used in BLAST to perform the initial all-pairs alignments. This results in a speedup of from 80x to 190x over the CPU code (8 cores). The quality is comparable to the original according to a commonly used benchmark suite evaluated with respect to multiple distance metrics.

The challenge in FPGA-based acceleration is finding a suitable application mapping. Unfortunately many software heuristics do not fall into this category and so other methods must be applied. One is restructuring: an entirely new algorithm is applied. Another is to analyze application utilization and develop accuracy/performance tradeoffs. Using our prefiltering approach and novel FPGA programming models we have achieved significant speedup over reference programs. We have applied approximation, seeding, and filtering to this end. The bulk of this study is to introduce the pros and cons of these acceleration models for biosequence analysis tools.

1 Introduction

1.1 The Problem

Bioinformatics refers to the analysis and management of scientific data and to the development of tools and applications that help us organize, retrieve, and process biological knowledge bases [Dur98][Jon04][Ewe05]. The application of mathematics and computer science for the modeling of biological processes has been essential to the use of biotic information for fundamental applications such as understanding life processes and in high impact applied domains such as drug discovery [Ach07][Jon04].

The key insight in bioinformatics is that biologically significant polymers, such as proteins and DNA, can be abstracted into character strings of a finite alphabet [Dur98]. Another fundamental observation is that all living cells pass a massive amount of hereditary features onto their offspring through a process of replication and cell division [Alb02]. In other words, nature adapts new sequences from pre-existing sequences. This opens the door for understanding the functionality of newly discovered sequences: by comparing a new sequence with known sequences, we can usually detect similarities that will help us learn about the structure and infer the functionality of that sequence. This mechanism allows biologists to use approximate string matching (AM) to determine, for example, how a newly identified protein is related to previously analyzed proteins, and how it has diverged through mutation [Mah10].

While AM is critical in diverse fields, e.g., text analysis, certain properties of biological sequences have required creation of biology-specific algorithms. Here the canonical AM task is Sequence Alignment (SA). For example, Hamming distance, the number of differing characters, is one way to measure differences between two strings, but does

not tolerate insertions or deletions (indels). As discussed later, more generalized scoring is necessary and is most often based on the probability of particular character mutations and includes indels; it can be handled using dynamic programming (DP) techniques. These have complexity O(mn) for two strings of size m and n, respectively. With the exploding size of biological databases, however, DP algorithms have often proven to be impractical. This has spawned heuristic O(n) algorithms, the most famous and widely used of these is BLAST [Alt90].

Since the completion of the human genome project, the scientific community has seen a sharp and rapid growth in the size of publicly available genomic and biotic information. Due to advances in technology and computing power, biological data are being produced at an exponential rate; genomic databases now double in size every 15 months [Ben12a]. The complexity of bioinformatic tasks to which sequence alignment is being applied is increasing just as dramatically. A typical query, say, of a protein with respect to a database of all other known proteins, requires millions of pairwise SAs. In Multiple Sequence Alignment (MSA), algorithms often begin with all-to-all pairwise sequence alignment. And Phylogenetic Analysis can require millions of MSAs. As a result, the development of faster SA tools and methods continues to be one of the fundamental challenges in Computational Biology.

Since its invention, BLAST has been based on heuristics [Alt90][Tho94] and algorithmic development remains an active area of research [Hen10][Hom09][Ken02]. On the other hand, the acceleration and parallelization of these applications are as important as algorithmic improvements. For example, the National Center for Biotechnology Information(NCBI) maintains a BLAST server, that consists of thousands of nodes that

serve the biological community. But while this valuable resource is sufficient for basic database searches, there remains a huge demand for complex and large-scale applications. For these acceleration is highly desirable.

Acceleration refers to the use of compute devices other than standard CPUs to speed up a computation. There are several ways to accelerate an application, the most popular of which currently is the application of GPUs. But in the case of bioinformatics, FPGAs have proved to be an excellent match and have often shown superior performance [Zou12][Ben12a]. ***The purpose of this study is to analyze and develop new methods for the application of FPGAs in order to accelerate standard SA and MSA tools.***

FPGAs are off-the-shelf integrated circuits that can be programmed by the user to perform a specific functionality [Sco10]. The critical challenge in FPGA-based acceleration is finding a good application mapping that is suitable for hardware implementation. Unfortunately, the heuristics applied in parallel application development often do not transfer to the FPGA. As a result, it is often necessary to restructure the program or to compromise accuracy for the sake of speedup.

We have developed a number of methods based on *prefiltering* [Mah10], [Mah12a],[Mah12b]. This method works outside the target application to quickly reducing the original workload, by 99.99% in the case of BLAST, while retaining the essential problem information. The target application then executes the remaining problem and obtains the correct answer in a fraction of the time of the original unaccelerated application. The advantage of this approach is that it leads to the compact implementations necessary to get high utilization of the FPGA while not

sacrificing correctness. This study serves to introduce the pros and cons of this acceleration method for FPGAs as applied to biosequence analysis tools.

1.2 Sequence Analysis Algorithms

The purpose of biosequence analysis is to find the relationship between known and (potentially) unknown sequences. This helps in discovering their functionality, features that contribute to their functionality, or the evolutionary relationship between multiple sequences. Sequence analysis algorithms can be categorized in many dimensions. Here, we list the most impotent categories:

- Pair-wise vs. multiple sequence alignment : pair-wise tries to optimally align two sequences, MSA tries to find the optimal alignment of multiple sequences [Smi81][Tho94].
- Gapped vs. ungapped alignment : Gapped allows indels in query and subject sequences, ungapped do not allow indels [Smi81][Har07]
- Local vs. global alignment : Global alignment have to align all characters in the two sequences, local alignment do not have this restriction [Smi81][Nee70].
- Optimal vs. suboptimal solutions [Smi81][Ach07].

There are many sequence analysis (SA) tools. Of these tools, the optimal solutions use some variation of dynamic programming (DP). As already described, however, this optimality is often not sufficiently important to compensate for their relative slowness compared with heuristic methods. In particular, BLAST is the dominant SA application; and of the many BLAST implementations, NCBI BLAST has become the de facto standard. In fact, biologists tend to ignore any application that deviates from this tool. It

is sometimes even assumed to be more accurate than the optimal DP methods because it removes the junk similarities – similarities with no biological root – from the final report. In this work, we will accelerate NCBI BLAST, the most widely used and one of the most highly optimized sequence alignment application.

A typical use-case starts with a query sequence. NCBI BLAST compares the query sequence to a database containing millions of subject sequences. It returns the most similar sequences alongside the best alignments. NCBI BLAST returns almost identical results as DP methods and is much faster.

Multiple sequence alignment (MSA) is the extension of pair-wise sequence alignment to multiple sequences [Gus97]. In a typical use-case, the user is interested in finding the relationship between thousands of sequences; i.e., finding their common ancestor or commonalities. The optimal MSA can be found with multidimensional DP, but, because the time and space complexity of multidimensional DP grows exponentially with the number of sequences processed, it is impractical and is almost never used. Once again, heuristics help. The heuristic approach often used in MSA is called progressive sequence alignment. It consists of a number of phases details of which are provided in the subsequent chapters. In this work, we will focus in acceleration of CLUSTALW, one of the most commonly used MSA applications.

1.3 High-Performance Computing with Accelerators

Traditionally, high-performance computing systems were considered to be either multiprocessing systems or massively parallel processing systems. These systems incorporate multiple identical CPU nodes in order to speed up a task. The cost-effective

use of these systems remains a challenging task. Memory bandwidth limitations and routing network congestion exacerbate the problem. In recent years, accelerator-based high-performance computing that exploits application-specific accelerators has gained more attention. The reason for this is that this method of computing allows a system to deliver more speedup with more flexibility in the programming interfaces and less power consumption than the traditional clusters. There are many accelerator based approaches. In brief, we review some of the alternatives that are currently available:

- Multicore: Multicore CPUs are now used everywhere. They are considered the simplest approach to speeding up applications. They can deliver impressive speedup if they are not limited by limitations posed by IO or an application's inherent serial nature.
- Cell processor: Cell processors utilize a single CPU and many vector processors. They can deliver a high degree of performance in many multimedia and vector-processing applications [Che07]. Nevertheless, they are considered a challenging environment for software developers.
- GPUs: Graphics processing units (GPUs) have been used to accelerate a variety of applications [Lin10][Lip88][Nic10]. They are commodity processing units that are found in every computer. They consist of thousands of simple processing elements and are suitable for applications that can benefit from parallel floating point executions.
- FPGAs: FPGAs are off-the-shelf hardware accelerators that can be programmed by the user. We will describe these accelerators in more details in the next section.

These solutions vary in power consumption, ease of use, time needed for development, computational capacity and programming models, and cost. All of these options are constantly used in a variety of applications, and their hardware design and underlying technology is constantly updated. There is plenty of research being conducted to show the suitability of each one of these options for a certain application or problem set. Nevertheless, there is no single consensus platform. Our work studies a large portion of the high-performance computing platforms for a specific application families.

1.4 FPGA-Based Accelerators

Currently, field programmable gate arrays (FPGAs) are used to accelerate hardware as a basic block in reconfigurable computing-based high-performance computing. The first modern-era FPGA was developed 30 years ago. Since then, FPGA technology has seen many advancements that have made them one of the best acceleration platforms [Awa09][Don12]. Although early FPGAs consisted of just a few configurable lookup tables and IO pins, modern high-end FPGAs consist of:

- Hundreds of thousands of reconfigurable lookup tables
- Hundreds of thousands of reconfigurable communication paths
- Thousands of block RAMs
- Thousands of DSP blocks
- Thousands of configurable IO pins

In addition, FPGA vendors provide hundreds of IP cores and interface modules that simplify programming of and interfacing to the FPGA. Moreover, FPGAs are one of the drivers of IC processing technology and follow Moore's Law in parallel with CPUs. This

means that each new generation of FPGAs nearly doubles in capacity. Thus porting existing designs to a new generation of FPGAs can immediately boost the performance of the system.

1.4.1 Programmability

FPGAs were initially used for rapid prototyping ASIC designs. The reprogrammability features of FPGAs are extremely helpful in the test process and development cycle. To save time and money, many ASIC developers test their design on an FPGA before porting it to a die. On the other hand, in comparison to other acceleration engines that are based on multithreading (multicore, GPU, and cell systems), FPGA development is much more challenging. For example, when mapping an application to a multicore CPU, the application developer should consider how to parallelize its application on the available cores efficiently. When mapping the same application to a FPGA, the programmer not only needs to know how to parallelize the code but also how to map the resulting solution to the hardware. Architectural decisions can have huge impact on the final result. Also, designing high-quality FPGAs that efficiently take advantage of the available resources on an FPGA requires an experienced designer. Hardware descriptions languages, in addition to accurate simulation and CAD tools, are used to hide some of the low-level details of the implementation. Nevertheless, the programmer often faces an expansive set of considerations to explore before writing the HDL code. Some of these considerations are as follows:

- Arithmetic precision and mode: For example, fixed-point vs. floating point arithmetic

- Algorithmic choices: For example, directed calculation vs. FFT
- FPGA interface mode: For example, streaming data vs. random memory access
- Pipelining: How efficient each submodule should work
- Replication: How many units of each sub unit is required to a have balanced system
- Reusability: If/how to use the existing IP cores
- Latency vs. throughput requirements
- Mapping decisions: Whether to use block RAMs or lookup tables to store specific data
- Acceleration approach; For example, filtering a large database vs. direct mapping
- Architectural decisions: Which portions of the code can/should be mapped to the FPGA and which portions should be run on the host CPU
- Modifications in data structures: How efficiently a reference data structure can be mapped to an FPGA, what kind of modifications are required to take advantage of the parallelism in an FPGA
- Mapping limitation: How well an algorithm is mapped to FPGA, for example, whether it causes routing congestion?
- Memory bandwidth requirements and IO overhead
- Testability: Arguably the most important factor of all

In case of sequence alignment, of all of these factors, only arithmetic choices are straightforward. All of the other factors can play a big role in the final result.

1.4.2 FPGAs for High-Performance Computing

Performance gain from FPGA acceleration is based on three factors: continuous payload delivery, parallelization, and pipelining. Together these can combine to compensate for the FPGAs' low operating frequency.

- Constant payload delivery: In contrast to CPUs, FPGAs generally do not need to process indexing and other "overhead" instructions. Most applications are designed to so that each function unit produces payload at every clock cycle.
- Pipelining: Pipelining is another form of parallelism. Pipelined hardware executes multiple instructions simultaneously. Because FPGAs are reconfigurable, the programmer can create a custom pipeline, often with 50-100 or more stages.
- Parallelization: Inside FPGAs, functional units can be replicated in order to increase performance.

In addition, because FPGAs work at a lower frequency, their power consumption is the lowest among all acceleration engines.

FPGA-based high-performance computing has its own limitations, such as:

- Chip-area limitations: Each FPGA has a limited amount of resources. With the increasing complexity of applications, it is rarely possible to map an entire application to an FPGA. This is exacerbated by the fact that complex memory access patterns create significant area overhead.

- Designer-expertise limitations: Working with FPGAs requires professional knowledge and experience. Even in the best case, FPGA designs often require a substantially longer design time in comparison to pure software solutions.

1.5 High-Performance Reconfigurable Computing for Sequence Analysis

This study of reconfigurable computing for bioinformatics is significant for two reasons. The first is the importance of the production applications we are trying to accelerate: speeding them up will enable more basic science to be performed. The second is the exploration of the design space for the FPGA-based acceleration of SA. This both reveals inherent challenges in using FPGAs for SA but also shows the potential performance gain that one can expect from the FPGA-based acceleration of SA tools.

The challenges of FPGA-based acceleration of SA are as follows:

- With regards to implementation: Sequence analysis tools utilize many heuristics to speed up the analysis task. Often these heuristics are tailored to better software implementation without any hardware considerations. Typically, they have an irregular data access pattern, which makes IO architecture a big challenge. The designer must implement a variation of the heuristics in the FPGA without losing agreement with the reference code. The designer must also be able to replicate and parallelize his code in order to gain performance. Other important considerations are how to parallelize the code and how to replicate it so that the available CAD tools can map the design efficiently to the target FPGA.

- With regards to performance: A production-level multithreaded SA code that runs on a high-end CPU with 3 GHz is already highly optimized and efficient. Accelerating such a code requires very careful design. From the hardware point of view, one should be able to take advantage of all the resources on the FPGA. The design units should be small so that they can be replicated. Also, the design units should be highly efficient and reasonably pipelined. In the pipelined architecture, there should be no load imbalance. From the software point of view, there should not be significant overhead in communication or in the reformatting and preparation of data structures.
- With regards to accuracy: Most of the time, when accelerating biosequence analysis tools, the designer does not have the luxury of losing selectivity to gain performance. That means that the computations must either be exactly mapped to the FPGA or that the emulation hardware should be strictly more sensitive.

The purpose of this study is investigate novel solutions in dealing with the challenges mentioned above. There are a number of previous studies that have tried to accelerate different SA tools on FPGAs. We will enhance these by proposing new approaches and investigating the design space.

1.6 Summary of Contributions

Two applications are accelerated in this study NCBI BLAST and CLUSTALW. These are, respectively, the most commonly used sequence alignment and multiple sequence alignment tools. For both applications we use prefiltering. At the end of this study, we present an analysis of the prefiltering approach as an acceleration mechanism.

1.6.1 Acceleration of NCBI BLAST

NCBI BLAST has become the de facto standard in bioinformatics approximate string matching and, as already described, its parallelization and acceleration are of fundamental importance. For example, massively parallel servers for BLAST have been constructed with the Blue Gene/L [Ran05]. Also, NCBI maintains a large server that processes hundreds of thousands of searches per day [McG04]. For generic clusters, mpiBLAST is one of the most popular of several parallel BLAST algorithms [Gar06]. FPGAs have probably been the most popular tool for the acceleration of NCBI BLAST, with commercial products from TimeLogic [Tim10] and Mitrionics [Mit10] and several academic efforts [Her07][Jac08][Lav06].

Public access to NCBI BLAST is possible either through the download of code or directly through a large web-accessible server. This standardization motivates the design criteria for accelerated BLAST codes; i.e., users not only expect performance to be significantly upgraded but also that outputs will exactly match the inputs given by the original system. BLAST implementations run through several phases and return some number of matches with respect to a statistical measure of likely significance.

The problem is that NCBI BLAST uses complex heuristics that make it difficult to simultaneously achieve both substantial speed-up and exact agreement with the original output. There are several approaches to accelerate NCBI BLAST. One approach is to profile the code and accelerate the most heavily used modules. This can give an agreement of outputs but may not achieve any performance gain, given that there are many paths that add up to bog down execution time. Accelerating enough of these paths may not be a viable solution, especially on an FPGA where code size translates

to chip area. A second approach is to restructure the code, modifying or bypassing some heuristics. This can lead to excellent performance but is unlikely to yield agreement. Academic FPGA-accelerated BLASTs [Her07][Jac08][Lav06] have mostly followed one approach or the other. The methods used by the commercial versions are typically either not publicly available or follow an academic version [Tim10][Mit10].

In this work we use a third approach : prefiltering. The idea behind prefiltering is to quickly reduce the size of a database to a small fraction and then use the original NCBI BLAST code to process the query. Agreement is achieved as follows. The prefiltering is constructed to guarantee that its output is strictly more sensitive than the original code; that is, no matches are missed but extra matches may be found. The latter can then be (optionally) removed by running NCBI BLAST on the reduced database. The primary result is a transparent FPGA-accelerated NCBI BLASTP that achieves both output identical to the original and a factor of 12x improvement in performance. The mechanism is the primary intellectual contribution of this work and consists of three highly efficient filters. The first implements two-hit seeding, the second performs exhaustive ungapped alignment, and the third performs gapped alignments. Furthermore, compared to a previous implementation of seeding heuristic, we have improved the accuracy of the two-hit seeding implementation. Also, we have improved the architecture of the exhaustive ungapped alignment filter to a degree that it is, now, orders of magnitude faster than a naive implementation.

1.6.2 Acceleration of CLUSTALW

Multiple sequence alignment is a critical tool for extracting and representing biologically important commonalities from a set of strings. While pair-wise sequence alignment is

used to assign possible functions to a protein, MSA goes to the next level. Among its uses are the prediction of function and secondary (two- and three-dimensional) structure, identification of the residues important for specificity of function, creation of alignments of distantly related sequences, and revealing clues about evolutionary history [Dur98]. While SA is typically used in database searches (finding correlations of one sequence with millions of anonymous candidates), MSA is generally applied to some number of sequences that are already hypothesized to have some commonality. And though it is often the case that some sequences are better understood or more important than others, MSA is basically an all-to-all matching problem. Another difference is that, while there is a consensus on the evaluation of pair-wise sequence alignments, on the basis of Karlin-Altschul statistics, with MSA, there is no objective way to define an unambiguously correct alignment [Dur98]. Therefore, evaluating MSA applications requires either expert knowledge or its surrogate through preselected sets of related sequences and encoded evaluation metrics.

In an MSA workflow, a number of sequences (k) of length n are aligned. The median value for n is about 300, but it is often closer to 1,000; k can range from a few to a few thousand sequences. Optimal MSA algorithms have been created by extending dynamic-programming-based SA to higher dimensions. These are exponentially complexity $O(n^k)$. Applying restrictions like those in [Ben12b] and [Liu11] results in tremendous speedups, making it plausible for k up to small double digits. A larger k, however, requires the use of heuristics such as progressive refinement [Fen87]. These codes typically run in three phases: (1) an all-to-all phase where all pairs are aligned and scored, (2) a tree-building phase where a guided tree is built that has sequences as

its leaves and whose interior nodes represent alignment order, and (3) a final phase where all pairs of nodes are aligned.

The most commonly used MSA code is CLUSTALW [Tho94]. When the FPGA-based DP method is ported to updated FPGAs and multicore CPUs, the speedup occurs in a similar range, but with some variance; i.e., from 18× to 58×. We use a different approach in creating a CLUSTALW-based FPGA-accelerated MSA (FMSA). Just as BLAST applies multiple passes of heuristics to emulate DP-based SA, so we apply BLAST-inspired filters to the pair-wise alignments. In particular we use a 2-hit filter (seeding pass) [Jac08] followed directly in a pipeline by an ungapped alignment (ungapped extension pass) [Mah10, Her07]. For the latter we emulate the ungapped mode of NCBI BLASTP.

There are two versions of FMSA, fast (FMSA-f) and emulation (FMSAe). In both cases, we use a scoring function analogous to the one used by CLUSTALW; i.e., rather than returning an E-value, FMSA computes a function based on identity counts. In fast mode, these scores are sent directly to the second phase of CLUSTALW to complete the processing. In emulation mode, some fraction of the high-scoring pairs are rescored with the DP-based method of Oliver et al. [Oli06] that emulates the CLUSTALW scoring function precisely. The result is a factor of from 80× to 189× speedup with respect to an eight-way parallel CPU code. The quality is comparable to the original according to a commonly used benchmark suite evaluated with respect to multiple distance metrics.

1.7 Organization of the Rest of the Thesis

The rest of this thesis is as follows:

Chapter 2 presents an overview of high-performance computing. It describes the methods used to accelerate different applications with the use of FPGAs, GPUs, and modern processors. It also presents a brief review of cluster computing.

Chapter 3 describes SA and MSA algorithms in detail. It describes the fundamental ideas in biosequence analysis, classic algorithms based on DP, and standard heuristic algorithms that are widely used; i.e., NCBI BLAST and CLUSTALW.

Chapter 4 presents a survey of previous attempts to accelerate SA methods. It includes all the related work in acceleration of NCBI BLAST, CLUSTALW, and Smith-Waterman.

Chapter 5 presents our FPGA-based accelerated NCBI BLAST, CAAD BLAST. It includes a detailed description of our seed-generation system and filtering approach. It presents several optimizations that significantly improve the performance of the final hardware-accelerated BLAST. It details our implementation on two different acceleration boards with two different mapping approaches: multiphase and pipelined. In addition, it provides a scalability analysis on different target FPGAs.

Chapter 6 presents our FPGA-accelerated CLUSTALW, FMSA. We have used an FPGA to accelerate CLUSTALW in both the emulation and fast mode. Using these two modes, we present a tradeoff analysis of speedup gain versus accuracy. We also present the speedup results over the reference code.

Chapter 7 concludes this thesis and provides guidelines and future work for the acceleration of SA tools.

2 High-Performance Computing

2.1 Overview

The word “supercomputing” refers to the fastest computing models available at each time; the computing models which provide the highest throughput and the lowest latency [Cul97]. The need for faster computers is always growing. A variety of scientific and industrial applications benefit from the high speed of high-performance computing [Gok05] [Cul97]. These applications include market analysis, climatology, computation biology, physics, and many more [Don12]. As an example, the newest generation of DNA sequencing machines [Hen10] produces massive amounts of data in a very short amount of time. For instance, one of the main goals of this approach is to provide the possibility of treatment based on personalized medicine; i.e., using medicine that is tuned to a specific patient's genetics [Met09]. In order to achieve this goal, the huge amounts of genomic data that are produced by these next-generation sequencing machines should be aligned to existing references and analyzed [She08]. This should be done in the shortest possible time, and this is where high-performance computing applied to SA can play a basic role.

The speed of computers has increased massively over the past century thanks to the increase in transistor count on chips over time, a phenomena known as Moore's Law. Nevertheless, the need for even faster computing resources still exists and will probably last as long as computers exist. This is generally due to two main factors: (1) the amount of raw data that is generated over time increases with the speed of computers

(i.e., the faster computers are, the more we can generate data with them) and (2) the complexity of the applications working on these data increases with computing power.

High-performance computing is a broad topic that includes many concepts in computer science and engineering, such as parallel computing, parallel hardware architectures, routing architectures, memory hierarchy, cluster computing, and custom processing units. A recent approach in high-performance computing is based on using non-microprocessor compute units such as FPGAs and GPUS [Gok05][Nic10]. The architecture of these systems can consist of any collection of GPUs, FPGAs, or custom ASIC accelerators which are used either singularly or as a cluster of computing resources. For example, the NOVO-G supercomputer consists of 296 top-end accelerator FPGAs, 26 Intel quad core Nehalem Xeon processors, and 576 GB total RAM [Geo11].

The rest of this chapter is as follows. In Section 2.2, we provide a background of high-performance computing. Sections 2.3 and 2.4 introduce multicore and GPU computing, respectively. Sections 2.5 and 2.6 provide a review of FPGAs and FPGA-based accelerators, respectively.

2.2 Background

The traditional classification of high-performance computers is based on Flynn's taxonomy of computer architectures [Cul97]. In general terms, this taxonomy classifies two dimensions in parallelism: instruction and data. As a result, Flynn’s taxonomy categorizes high-performance computers into four groups [Don12]:

1. SISD, in which a single stream of data is processed by a single processing unit. For example, the traditional single-core PC, which executes a sequential serial code.
2. SIMD, where, at each cycle, a single instruction is executed on multiple data streams in parallel. An array processor is a well-known example of this type of computer. An SIMD instruction set is an instruction set that supports this type of processing. A well-known example is Intel's SSE extension [Int11].
3. MISD, where multiple instructions that related to a single data item are executed in parallel. This category subsumes many fault tolerant hardware techniques.
4. MIMD, where multiple instruction streams are executed in parallel and each instruction stream consumes its own data streams. A well-known example of this type of architecture is the contemporary multicore superscalar CPU [Pat90].

From another point of view, one can categorize supercomputers into two groups: shared memory systems and distributed memory systems [Cul97].

In a shared memory system, all the processing units have direct access to a main system-wide memory. The main idea behind these systems is that the processing units have equal access to the main memory, and, consequently, the memory transactions generated by multiple processing units can be handled transparently and evenly [Don12]. As a result, the programmer does not need to consider the location of the data on the system and does not need to worry about the efficiency of accessing a certain data item.

In a distributed memory system, each processing unit has its own local memory [Cul97]. In a processing system with a distributed memory architecture, each processing node consists of one or many processing units, each with its own local memory. In order to provide a node access to another node's local memory, the nodes are interconnected by network topology. Since, in a distributed memory system, the total provided memory bandwidth has a direct relationship with the number of processing nodes, these systems have a clear advantage over shared memory systems with regard to memory bandwidth and its scalability. Furthermore, in these systems, the speed of each memory is of less concern in comparison to shared memory systems. On the other hand, distributed memory systems have their own disadvantages. In comparison to shared memory systems, in distributed systems, the communication and synchronization overhead between distributed nodes is higher. Thus, it is possible that the running speed of an application on this type of systems can suffer from the creation of inter-node communication bottlenecks [Don12].

A computer cluster is a set of loosely connected computers that work together to the extent that, in many respects, they can be viewed as a single system. Since the introduction of the Beowulf cluster in 1994, computer clusters have become widely used and commercialized. This has been mainly because of their relatively low cost, their ease of engineering, and their simple setup process [Don12]. The structure of a cluster usually follows a client server computation model. Often, the computers are connected via a local area network. Typically, a cluster consists of one or a few server nodes and lots of client nodes. In these systems, a special middleware software is often run on top of the operating system. This middleware software orchestrates the operations of

clients, dispatches the tasks to clients, and retrieves and organizes the results. In order to provide communication between cluster nodes, several programming approaches have been used. The two common approaches used in computer clusters massage passing interface (MPI) and private virtual machine (PVM). The increased power consumption, in combination with the existing limitations in the total size and volume of these systems, makes the scalability of computer clusters a big challenge.

In recent years, clusters of high-performance computing nodes that exploit application-specific hardware accelerators, such as FPGAs and ASIC, have gained popularity [Geo11]. Factors that have contributed this popularity gain include lower power consumption, increased flexibility, increased capabilities, significant speedup gains, increased debugging and testing capabilities, and the fact that upgrading to a new technology level can be easily handled with the existing programming environments and CAD tools.

From here, we will give an overview of common high-performance computing systems based on custom accelerators, but, before that, we will take a look at current multicore processor technology. The TOP500 lists the 500 most powerful computing systems in the world [Top13].

2.3 Multicore Processors

Over time, processors have increased extensively in capacity. Increasing chip density has allowed the extraction of more instruction-level parallelism. The performance of microprocessors has improved steadily over time because of increasing transistor count and operating frequencies. However, in the past decade, the performance of a single

processor has reached a plateau. Issues like energy consumption and heat dissipation limit operating frequency to about 4 GHz. At this point, the architecture of a single core hardly benefits from an increase in transistor count. In other words, due to energy consumption and heat dissipation problems, the operating frequency and complexity of processors have hit a so-called performance wall.

Since 2003, processor vendors have taken a different approach to increasing the computing power of processors. This new approach mainly involves integrating multiple processor cores into a microprocessor and introducing multicore CPUs.

This has caused a revolution in the way efficient programs are written. Nowadays, most programs benefit from potential performance gains of multithreading. The era of sequential programming on a single-core CPU has reached an end, and a new interest in parallel programming has begun with the so-called concurrency revolution [Olu05].

For instance, a 45nm Intel Nehalem Ex processor has eight cores per CPU working with 2.91 GHz clock. It has an aggregate peak memory bandwidth of 43 GB/s and 10 G/s per memory channel. Several years ago, this microprocessor was considered a shared memory supercomputer.

Since there are multiple independent processing cores available on each microprocessor, a programmer can potentially and dramatically increase the performance of an application. This is usually done by means of implicit or explicit multithreading. There are several threading libraries available. The two most well-known threading mechanisms are PThreads and OpenMP.

With the improvements in the memory bandwidth, a carefully multithreaded code can achieve linear speedup over a single threaded code for many applications. Of course the theoretical limit will be the number of cores integrated into the CPU. For example we noticed this issue in the latest versions of NCBI BLAST. NCBI BLAST which will be introduced later in this report, is for the most part an embarrassingly parallel sequence analysis tool. The performance of this application, when run in multithreaded mode, simply scales linearly with the number of cores in the CPU.

2.4 GPU Computing

Originally, graphics processing units (GPUs) were developed for use as graphics-rendering engines. Nowadays, GPUs are also used as general purpose acceleration engines. General purpose GPU computing has become especially popular since NVIDIA introduced CUDA (compute unified device architecture), a C extension that enables applications to be ported to GPUs. Nevertheless, GPU programming for acceleration has its own limitations.

The processing power of GPUs has increased significantly over the past decade. The first NVIDIA GeForce 3 GPU series that was marketed in 2001 only had a four pixel pipeline, whereas a more recent NVIDIA Tesla GPU has up to 2,688 streaming processors [Lin08]. The driving force behind this massive evolution is the ever-growing demands in the game industry [Nic10].

A GPU consists of many simple floating point processing elements. In this way, a GPU is essentially a shared memory single instruction multiple thread computing platform.

Each processing element has a fully pipelined integer unit and a fully pipelined floating point unit. The threads are executed in groups of 32 called warps.

The following figure shows the internal architecture of an NVIDIA Tesla Geforce 8800 GPU [Lin08].

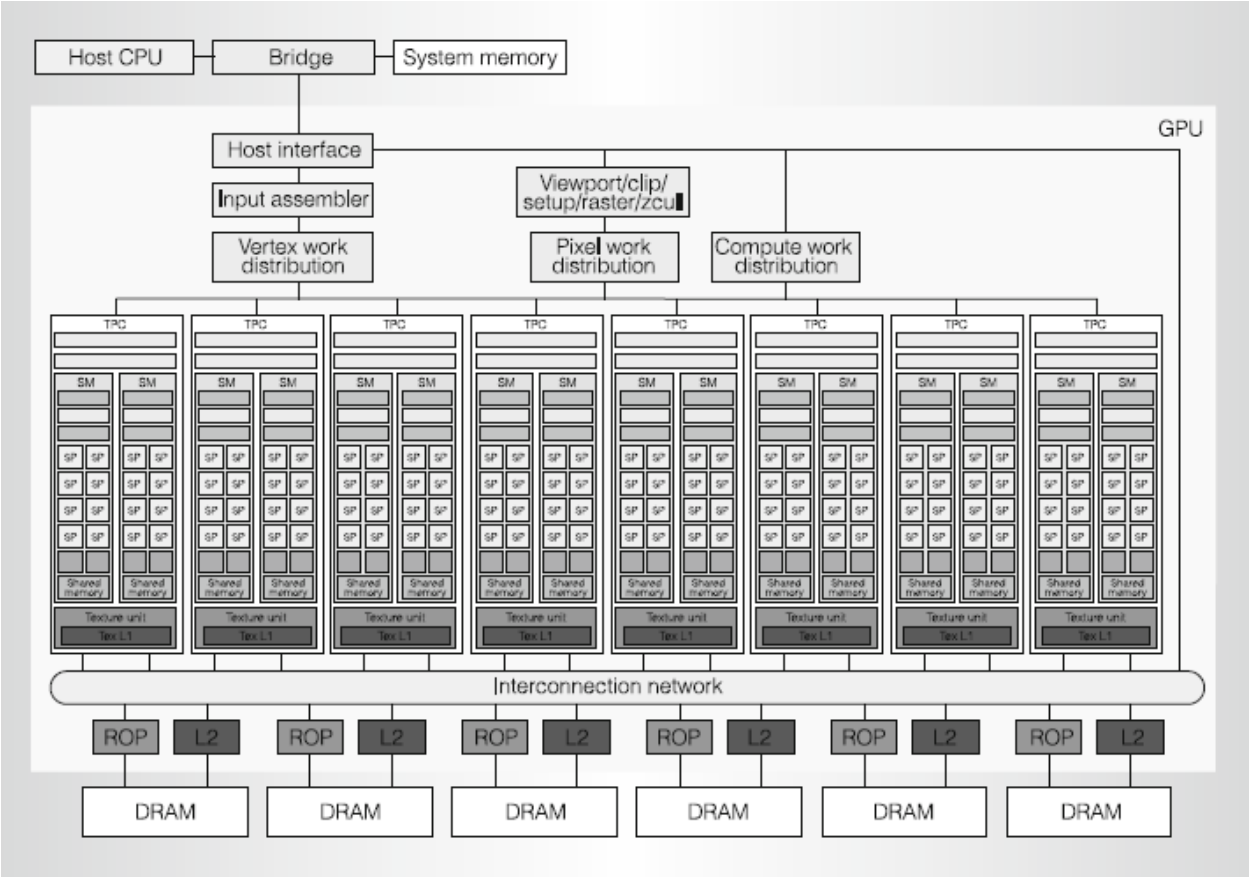


Figure 2-1 A Modern GPU Architecture [Lin08]

This GPU consists of 128 streaming processor (SP) cores organized as 16 streaming multiprocessors (SMs). The streaming processors are clocked at 1.5 GHz. Each streaming multiprocessor has 16 KB of shared memory. Shared memory is the fastest memory entity in the GPU memory hierarchy. The shared memory can be accessed by

all Streaming Processors in a Streaming Multiprocessor. It provides a fast and efficient way for threads of an SM to communicate and synchronize. Generally speaking, one can say that a shared memory system has a similar role as a cache in a traditional CPU, except that it is the responsibility of the programmer to use it efficiently; there is no automatic caching.

Each SM can execute up to 768 threads without any scheduling overhead. All threads in an active warp should execute the same instructions. If the threads diverge through branching, the scheduler will execute the branches serially. This, in turn, will reduce the performance and efficiency of GPU execution. Synchronization between threads that are scheduled to be executed on different SMs can be done through the global memory. This in particular can have a negative impact on the performance of the accelerated system due to the slower bandwidth of the global memory.

The CUDA programming model simplifies mapping applications to GPU architecture on the basis of data parallel problem decomposition [Nic10]. The programmer finds portions of the code that can be parallelized and decomposes the data array into a two-dimensional grid of thread blocks where each thread block, in turn, is a three-dimensional collection of threads. When a GPU kernel is called, each streaming multiprocessor executes up to eight thread blocks, depending on the recourse requirements of each thread. An active SM which has sufficient resources executes the thread blocks concurrently as warps of 32 threads.

Compared to other acceleration engines, GPU is more suitable for applications that show massive SIMD like data parallelism and require lots of floating point calculations.

The performance gain from GPU acceleration is thus largely moderate (e.g. 5x) and application dependant.

2.5 FPGAs

Field programmable gate arrays (FPGAs) are prefabricated integrated circuits that can be programmed by the customer after it is manufactured to become almost any circuit or system [Awa09]. The idea of programmable devices was introduced and developed in the 1960s with programmable logic arrays (PLAs), programmable array logic (PAL), and read-only memory (ROM). A PLA or PAL consist of a regular array of prefabricated gates with a programmable interconnect architecture. Figure 2-2 shows a PLA structures.

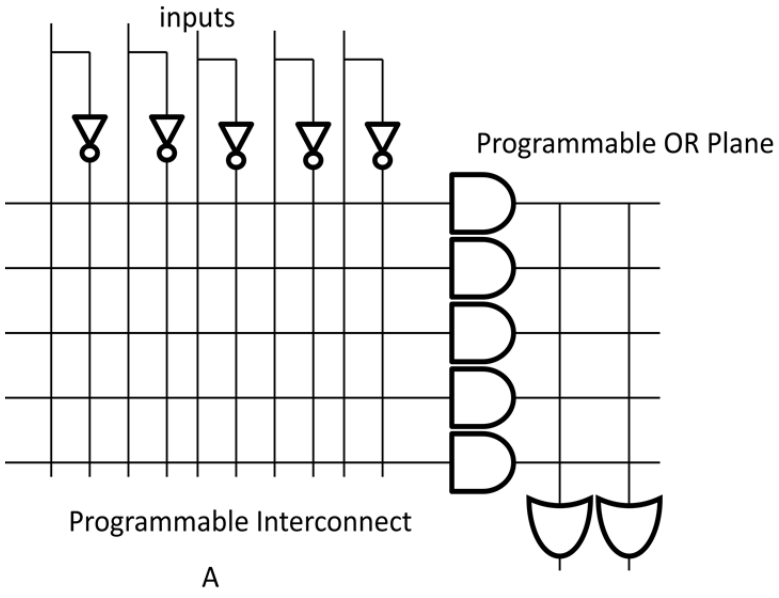


Figure 2-2 PLA example

These architectures only supported a very limited two-level and/or logic and suffered from significant routing overhead. Implementation of a realistic multilevel logic on these architectures was not possible because of prohibitive routing and area overhead.

The first modern era FPGA was developed by Xilinx in the 1980s. It was called XC2064, and it consisted of a number of programmable lookup tables and interconnects. XC2064 had 46 configurable logic blocks (CLBs), each consisting of two three-input lookup tables. The chip had only 58 I/O pins. Nowadays, almost 30 years later, a high-end Xilinx FPGA has more than 300,000 CLBs in addition to thousands of other high-end memory and DSP blocks [Xil13].

The initial market for FPGAs was mainly about prototyping integrated circuits in the development process of the application-specific integrated circuits. At that time, an FPGA was used as a less efficient and a demo version of the production level IC, so that the developer could have the chance to test and debug their circuit multiple times. Furthermore, a programmer could do this with much less cost and in a significantly shorter amount of time. Over time, though, FPGAs have evolved so much so that they have become a competitor in the ASIC market. Nowadays, compared with what they could have done before, FPGAs can deliver much higher performance. Interestingly, when compared to other acceleration engines and approaches, such as GPUs and clusters, FPGAs provide the biggest savings in power consumption other than ASICs themselves.

A state-of-the-art FPGA consists of a pool of programmable logic blocks, programmable IO blocks, configurable routing resources, several megabytes of memory block RAMs,

one or more embedded processing units, such as IBM PowerPCs, and an extensive set of commonly used DSP blocks, such as multipliers and adders. The programmable logic block implements the desired functionality, whereas the programmable interconnect allows the functional blocks to be interconnected as desired by the programmer. The programmable I/O connects the chip to the outside world on the basis of user settings.

The term "field programmable" means that a device can be configured after silicon fabrication. Thus, a field programmable device provides the possibility for a user to change the behavior of the device as needed. In order to provide programmability for FPGAs, three different methodologies, namely Anti-Fuse, EEPROM, and SRAM, have been used [Kuo08]. Over time, the methodology of SRAM-based programmability has grown to dominate other methodologies, and, nowadays, almost all commercial FPGAs use SRAM technology. There are several reasons for the widespread use of SRAM FPGAs. In contrast to the other two methodologies, SRAM-based technology provides infinite reprogrammability. While SRAM-based FPGAs use standard CMOS technology, other methodologies require technological capabilities beyond standard CMOS [Kuo08]. Furthermore, SRAM-based FPGAs are also easier to program and require no additional devices to program.

In SRAM-based FPGAs, in order to provide reconfigurability in logic blocks and interconnects, static memory cells are distributed across the FPGA. The most basic logic element of an FPGA is called a lookup table (LUT). As shown in Figure 2-3, an *n* input LUT consists of 2^n static cells and a $2^n{:}\,1$ multiplexer. The *n* inputs are connected to the multiplexer select lines and steer one of the 2^n static cells to the output. In an *n* input LUT, any *n* input logic function can be realized. This can be done by setting the

desired bits in the static cells of the lookup table. Nowadays, a typical FPGA has four, five, or six input lookup tables. A six input lookup table can implement any function of 6 bits.

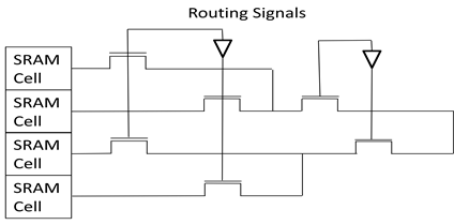


Figure 2-3 Static Memory Cells and Lookup Table [Kuo08]

Similarly, static cells are used to steer signals through the reconfigurable interconnects of the FPGA. As shown in Figure 2-4, the reconfigurability of FPGA interconnects is provided using multiplexers and static cells. At each junction on the FPGA routing mesh, a programmable switch based on multiplexers and static cells can connect any two lines to each other. The programmer connects two signals by setting the proper bits in the interconnects' configurable static cells.

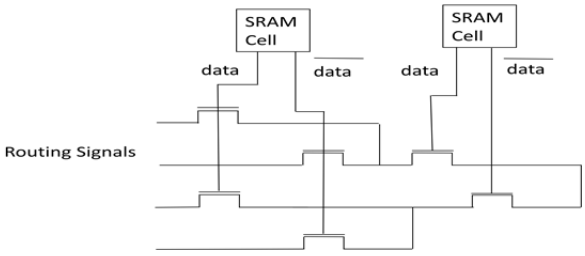


Figure 2-4 Multiplexer and Static Memory cell [Kuo08]

By itself, a LUT implements a combinational logic. In order to provide the possibility of implementing sequential logic, the output of the lookup table can optionally pass

through a flip-flop. A static cell configuration bit and a 2:1 multiplexor are used to provide this option.

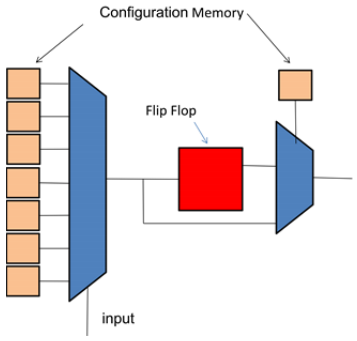


Figure 2-5 Sequential logic in FPGA

In addition to configurable logic blocks and interconnects, a modern FPGA consists of an array of independently addressable block RAMs and hundreds of hardwired DSP blocks along with one or two embedded processor cores. Similar to logic blocks, the inputs and outputs of these modules can be connected to any other module or logic block through the programmable interconnects.

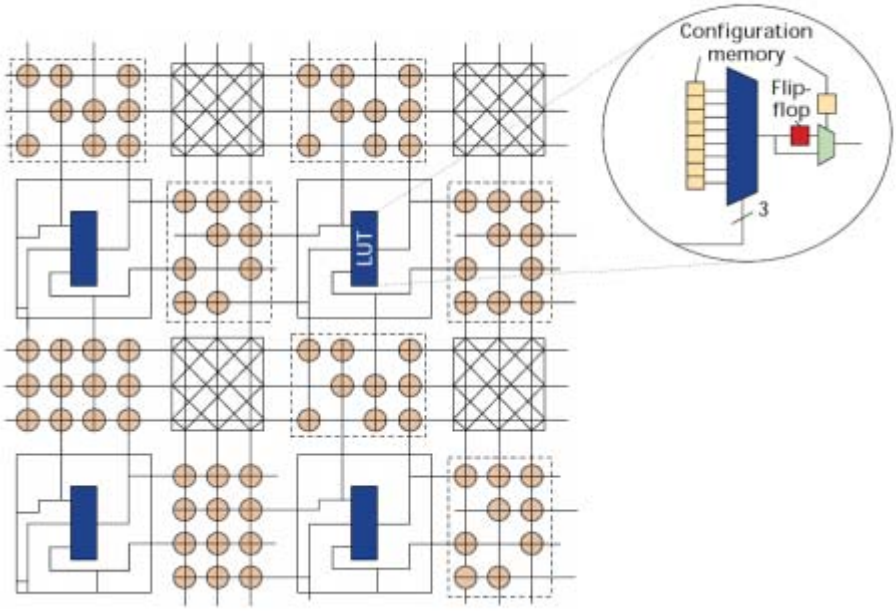


Figure 2-6 FPGA Interconnect Schematic view

At a higher level, the FPGA vendor supplies the necessary IP cores that can be used to interface with off-chip resources in a flexible and easy-to-use way and the software drivers to use these IP cores if hardware/software codesign is required.

For more than a decade, FPGAs have been used to accelerate a variety of tools and applications, including telecommunication and networking applications, signal and image processing, control systems, biomedical applications, and many other practical applications [Gok05].

The advantages of using FPGAs in high-performance computing can be summarized as follows:

- **Higher performance**: FPGAs often deliver the highest performance among all accelerators, particularly for low precision applications. A speedup gain of 10x to 300x using FPGAs is commonly seen in published literature. The main reason for this is the vast parallelism that can be achieved inside FPGAs by fine-grain pipelining, coarse-grain replication, and a huge amount of memory bandwidth from the block RAMs inside FPGAs. In addition, as Moore's Law remains valid, the resources in FPGAs increase over time, which increases the possibility for more parallelism and inherently more performance.
- **Lower power consumption:** FPGAs consume much less power than CPUs and GPUs because of their lower operating frequencies.
- **Reconfigurability**: The fact that FPGAs are reprogrammable gives them a big advantage over ASICs. Some applications require more frequent reprogramming. In either case, reconfigurability saves time and money.
- **Time to market**: The design cycle of FPGAs is much shorter than that of ASICs.
- **Technology upgrades**: Migrating a design from an old FPGA to a new FPGA requires little time and effort but can result in a significant gain in performance.

The challenges of FPGA based design include:

- **Higher price**: FPGAs are more expensive than GPUs and CPUs, and, thus, the expectations are higher.
- **Limited resources**: FPGAs have limited resources. It's the developer's task to efficiently use these resources in the most efficient way.

2.6 FPGA-Based Systems

In brief, we state our assumptions about the target systems of this study with FPGA-based accelerators. These systems are typical for current products. The overall FPGA-based system consists of some number of standard nodes. Typical node configurations have one to four accelerator boards plugged into a high-speed connection (e.g., the front side bus or PCI Express). The host node runs the main application program. Nodes communicate with the accelerators through function calls. Each accelerator board consists of one to four FPGAs, memory, and a bus interface. On-board memory is tightly coupled to each FPGA, either through several interfaces (e.g., 6 x 32 bit) or a wide bus (128 bit). Currently, 4 GB–64 GB of memory per FPGA is standard. Besides configurable logic, the FPGA has dedicated components such as independently accessible multiport memories (e.g., 1,000 x 1 KB) called block RAMs (or BRAMs) and a similar number of multipliers. FPGAs used in high-performance reconfigurable computing typically run at 200 MHz, although, with optimization, substantially higher-operating frequencies can sometimes be achieved. In this research we have used two FPGA based acceleration platforms: Gidel board and the Convey machine. Next, we briefly describe their architectures.

2.6.1 Convey System

A Convey HC-1ex computer is a hybrid processor with a single four-core Intel CPU (Xeon L5408 2.13 GHz) and four Xilinx FPGAs (Virtex-6 XC6VLX76) [Bak10][Con13a][Con13b]. There is a total of 24 GB of host and coprocessor memory, a standard Intel IO chipset, and a reconfigurable coprocessor based on FPGA

technology. The system runs 64 bit Linux. The coprocessors are programmed by the user and can execute custom instructions. The host and coprocessors share the same virtual address space.

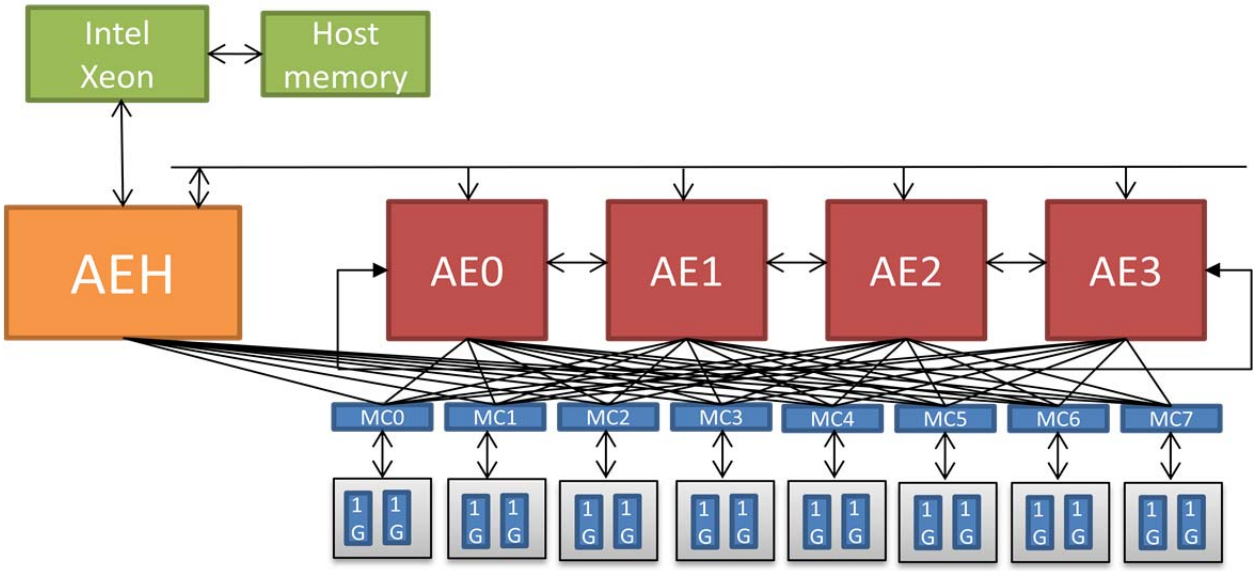


Figure 2-7 Convey System Overview [Bak10]

The user logic runs on 4 Xilinx Virtex 6 FPGAs, which Convey refers to as application engines (AEs). The coprocessor also consists of interface logic, called the application engine hub (AEH), which connects the coprocessors to the host CPU. It is responsible for fetching and decoding instructions, executing scalar instructions, and routing host memory requests to coprocessor memories. In addition to the AEH, the coprocessor system consists of eight memory controllers that connect the AEH and the AEs to coprocessor memory modules through a full crossbar network. The memory controller subsystem can support up to 16 DDR2 memory channels. The memory subsystem can collectively support up to 8,000 parallel requests and 80 GB/s total bandwidth.

In addition to user logic, each application engine includes some API logic that implements the interface between the application engine, the AEH, and the memory controllers. It also includes dispatch logic that enables the execution of the custom instruction and some management and debugging interface.

2.6.2 Gidel Board

The Gidel Proce III board is an FPGA acceleration board that connects to the system through a PCIex bus [Gid10]. The FPGA is an Altera Stratix-III 260E. For memory there, is 4.5 GB of DRAM partitioned into three banks of 2 GB, 2 GB, and 512 MB, respectively. Each bank has a 64 bit interface and can be accessed independently. One of the 2 GB and the 512 MB banks run at 333 MHz; the other 2 GB bank runs at 166 MHz. Data is transferred to and from the board by means of direct memory access (DMA) channels through the PCIex bus. The total DMA bandwidth can be up to 1 GB/s. The Gidel board provides a graphical user interface that is used to generate the hardware and software interface for the user logic and application. The following figure shows a block diagram of the Gidel board.

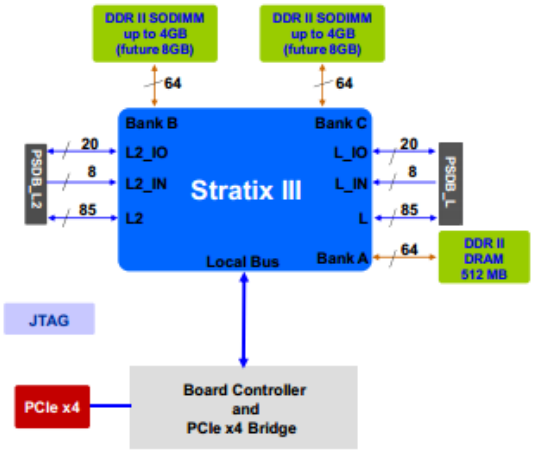


Figure 2-8 Gidel Board Overview [Gid13]

2.7 Summary

In this chapter we reviewed basic concepts in high performance computing. We took a brief look at the current status of multicore processors and GPU computing. We gave an introduction to FPGA architecture. We reviewed the possibilities of acceleration based high performance computing. At the end we introduced two acceleration platforms that we will use throughout the thesis Convey machine and Gidel board. Convey machine is a shared memory super computing platform consisting of 4 high-end FPGAs. The FPGAs can be programmed to accelerate applications using user defined custom instructions. Gidel board is a commodity acceleration platform that is connected to the system's PCI bus. Using vendor provided API and user interfaces the programmer can communicate with the FPGA (e.g. with a DMA call).

3 Sequence Analysis: Methods and Algorithms

3.1 Overview

A fundamental insight of bioinformatics is that principal biological polymers such as proteins and DNA can be abstracted into character strings (sequences). This allows biologists to use approximate string matching (AM) to determine, for example, how a newly identified protein is related to those previously analyzed, and how it has diverged through mutation. While classic dynamic programming methods can be used to this end, fast methods, such as BLAST, are based on heuristics, and can match a typical sequence (a query) against a set of known sequences (e.g., the millions in the NR database) in just a few minutes. Moreover, these heuristics only rarely miss significant matches. These remarkable results have only increased the importance of BLAST: it is now often used as the “inner loop” in more complex bioinformatics applications such as multiple alignment, genomics, and phylogenetics.

Multiple Sequence Alignment is critical to many bioinformatics solutions, e.g., in determining the structure and function of molecules from putative families of sequences in phylogenetics and finding the evolutionary relationship between species.

In this chapter we will look at the most important sequence analysis tools and algorithms. We will start with the basic biology of cell to give an insight into how sequence analysis comes into play. In Section 3.3 we will overview the fundamental concepts in sequence analysis. In Section 3.4 we will look at different scoring models used in sequence analysis. Section 3.5 provides an overview of pairwise sequence

analysis methods and algorithms including Smith-Waterman and NCBI BLAST. Section 3.6.5 details how statistical significance is determined in sequence analysis use-cases. Section 3.7 provides details of important multiple sequence alignment methods.

3.2 The Basic Biology of Cell

A fundamental feature of all living organisms is heredity [Alb02]. A living creature passes down heredity information to its offspring, specifying a massive detail of characteristics that its offspring should posses. All living organisms consist of cells. Regardless of the number of constituent cells, a living organism is generated by cell divisions from a single cell. Thus, A single cell, not only stores all the hereditary information in an organism but also has all the resources required to replicate itself. All cells depend on three principal molecules to function: DNA, RNA, and proteins [Alb02]. A cell's DNA contains the entirety of an organism's hereditary information. All living cells store their hereditary genetic information in double-stranded molecules of DNA, which act as a database of features. The four bases that make up a DNA strand are adenine (A), guanine (G), cytosine (C), and thymine (T). A DNA strand is often represented as a chain of nucleotides where each nucleotide consists of a sugar, a phosphate attached to it, and one of the four bases named above. The long chain of A, T, C, and G monomers of a DNA strand encodes the genetic information of the living cell that it belongs to. A single-stranded DNA molecule is extended by adding nucleotides to its ends. The added base can be any of the four bases, since there is only one sugar-phosphate backbone. However, in a double-stranded DNA, an A in one strand always bonds with a T in another; similarly, a C always bonds with a G. This way, during the

process of replication, a single strand of the DNA is used as a template by the cell to create identical copies. Because of this constraint in bonding, the term “base pair” is often used in literature. Since the bonds between bases are much weaker than the bonds between the phosphate and sugar constituents of the backbone, the two strands can be pulled apart without breaking the backbone. On the basis of the complementary bonding constraint described above, the two strands then act as a template to create two identical copies of the original double-stranded DNA [Alb02].

Just like DNA and RNA, proteins are long, unbranched polymers formed by chaining many monomeric building blocks. Just like DNA and RNA, the monomeric building blocks are the same for all proteins. On the other hand, the protein monomers that are called amino acids are very different from those of DNA and RNA. There are 20 amino acids, as opposed to the four bases of DNA and RNA. Thus compared with DNA or RNA strings which consist of 4 symbols, the alphabet of the proteomic strings consists of 20 symbols. Whereas RNA are considered the translators of the genetic code, proteins are considered its running engine. Thus, there is generally a functional relationship between a DNA sequence and a protein sequence. Each protein has its own genetic functionality that is specified by its sequence of amino acids.

3.3 Fundamentals of Biosequence Analysis

Bioinformatics is the application of computer science and information technology to the field of biology. The fundamental observation in bioinformatics is that biological entities like proteins and DNA can be represented as character strings. A DNA (or RNA) is a sequence made from repeating A, C, G, and T (U in RNA). Similarly, proteins can be decoded as finite sequences of 20 characters. The theory of evolution states that

species have evolved over millions of years through a process of incremental change. With the invention of genome sequencing, scientists have been able to describe the process of evolution using the genomic sequence analysis.

The information stored in a DNA molecule is the result of evolution over time. This information is passed from the parent cell to the child cell during the replication process. Although the replication process is delicately accurate, it can introduce changes in the DNA sequence. Many factors can affect the accuracy of the replication and introduce errors. Just like any transmission mechanism, these errors can occur in three forms: substitution, insertion, or deletion of a symbol from the target result. These changes can occur in DNA, RNA, or protein sequences. It is expected that two biological sequences that have many common residues, whether they are DNA nucleotides or protein amino acids, will exhibit similar features or play similar roles in the development and functionality of a cell. Thus, sequence analysis methods can be used to detect the relationship between different biological sequences, to find the functionality of the newly found genes or proteins, to discover new drugs, or to provide new insights in understanding life itself.

The most fundamental and routinely asked question in biosequence analysis is therefore how these sequences are related. In order to answer this question, the two sequences must be aligned, and the alignments should be evaluated with a biologically meaningful metric. Durbin et al. lists the key issues involved in sequence analysis as [Dur98]:

1) Scoring models used to align sequences,

2) Alignment type and methods

3) Statistical methods used to evaluate the alignment.

In the follows sections, we will take a brief look at each one of these issues.

3.4 Scoring Models

There are many ways to score the similarity between character sequences. The simplest way to assess an alignment is by using Hamming distance. Hamming distance assumes that the two sequences being compared are already aligned in order; i.e., the *ith* symbol in one sequence is aligned with *ith* symbol in the other. But biologists usually do not have the luxury of assuming in order alignment. Since genomic sequences are subject to insertions and deletions, Hamming distance is not often used in sequence analysis.

Another way to measure the similarity between two sequences is by using the so-called edit distance. The edit distance between two sequences is the number of edit operations that are required in order to transform one sequence into another. The changes can be the insertion of a symbol, the deletion of a symbol, or the replacement of one symbol with another. This is also referred as Levenshtein distance. Dynamic programming can be used to calculate the edit distance. In calculating this distance, matching residues score zero, and all mismatches, insertions, or deletions are penalized by one. Similarly, a weighted edit distance can be used. In a weighted edit distance, two different cost values are used: one for mismatches and one for gaps. Each insertion or deletion is penalized with *D*, and each mismatch is penalized with *R*. This simple scoring matrix is often used for DNA sequence analysis.

As explained before, several biological and chemical factors affect errors in the replication process. Using statistical methods and expert knowledge, biologists have developed scoring matrices that represent these factors. For example, changing *R* to *Q* is much more biologically plausible than changing *E* to *C*. In a scoring matrix, this can be represented by having a positive score for the R/Q pair and a negative score for the E/C pair. Then, an alignment of two sequences can be scored with a simple summation. For each residue pair that is aligned, the corresponding score is added to a total running score. The gaps can be treated as special characters. Additive scoring has proven to be the best scoring mechanism for this so far. The following figure shows an alignment and its score.

D	M	F	C	N	T	E	G	I	A
|	|	|	|	|	|	|	|	|	|
T	M	G	-	N	-	E	G	Q	S
-1	+5	-3	-9	+6	-9	+7	+5	-2	-1

Figure 3-1 Sequence Alignment

Statistically speaking, this (and related similar methods) assumes that changes in a sequence occur independently of each other. In other words, one change does not affect other changes. As a result, additive scoring is the prevalent scheme in the analysis of DNA and protein sequences and is used as a standard tool. The following figure shows blosum62, one of several standard matrices used in protein sequencing and alignment scoring [Hen92][NCBa].

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	4																							
R	-1	5																						
N	-2	0	6																					
D	-2	-2	1	6																				
C	0	-3	-3	-3	9																			
Q	-1	1	0	0	-3	5																		
E	-1	0	0	2	-4	2	5																	
G	0	-2	0	-1	-3	-2	-2	6																
H	-2	0	1	-1	-3	0	0	-2	8															
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4														
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4													
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5												
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5											
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6										
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7									
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4								
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5							
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11						
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7					
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4				
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4			
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4		
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	
*	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Figure 3-2 Blosum62 Matrix

Gaps need special attention. There are two common ways to score gaps: linear or affine gap penalty. In the linear gap model, a single gap costs a constant (*V*), and the total cost of a gap of length (*L*) is $L \times V$. In affine gap model, opening a gap costs more than extending it. Thus, a gap of length *L* costs $U + V \times (L - 1)$, where *U* is the gap opening penalty and *V* is the gap extension penalty. It has been shown that, in modeling biological sequence similarities, the affine gap penalty is more accurate than the linear penalty. The affine gap penalty is slightly more costly computationally—both in hardware and in software—than the linear gap penalty.

3.5 Pairwise Sequence Alignment with Dynamic Programming

Pair-wise alignment algorithms can be divided into two subcategories: global alignment algorithms and local alignment algorithms. A global alignment algorithm aligns all of the residues in one sequence to all of the residues in another one, possibly by inserting gaps in the sequences. On contrast, in a local alignment algorithm, it is not required to

include all of the residues of both sequences, and only portions of the sequences that align better are of interest.

From another point of view, an alignment algorithm can be categorized as gapped or ungapped. A gapped alignment algorithm allows the insertion of gaps into the (sub)sequences, whereas an ungapped alignment algorithm does not. An ungapped alignment aligns contiguous portions of the two sequences.

There are numerous algorithms for solving the approximate string matching problem, but only a few of them are used for biosequence analysis. In this and the following two Sections we will look at the alignment algorithms that are commonly used in the biosequence analysis community, starting with methods based on Dynamic Programming (DP) and continuing with the most commonly used method, BLAST.

As shown below, an alignment of two sequences can be depicted with a tableau with one of the two sequences placed on the horizontal axis and the other on the vertical axis. In this depiction, diagonal arrows represent replacement or matching pairs, whereas vertical or horizontal arrows represent indels. Contents of the tableau are the local running match scores.

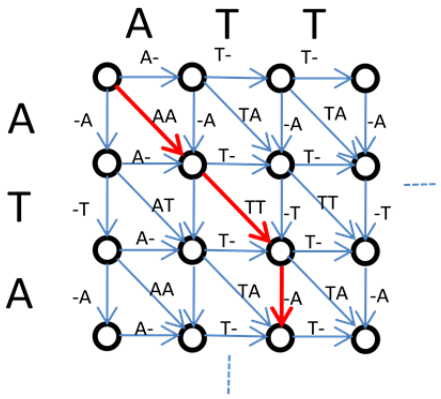


Figure 3-3 Alignment tableau

The problem of finding the best alignment between two sequences has all of the properties needed to make DP a suitable solution; that is, it has an optimal substructure and contains overlapping subproblems. DP recursion is often used to show the subproblem structure. Sometimes a tabular grid (similar to one shown in Figure 3-3) is used to show the DP solution. Each grid location corresponds to a subproblem of the problem of interest. The value written in each grid location represents the best score for the corresponding subproblem. In the case of sequence alignment, the subproblems are the scores of best alignments of the subsequences that are represented by grid locations.

A typical sequence alignment use-case starts with a query sequence and a database of known sequences. We call each sequence of this database a subject sequence. The sequence alignment tool aligns the query sequence with all of the subject sequences in the database, and those sequences that score high are returned alongside the optimal alignments that are found.

The two classic DP algorithms for approximate string matching are refereed as the Needleman-Wunsch and Smith-Waterman algorithms.

The Needleman-Wunsch algorithm [Nee70] is the classic dynamic programming algorithm for solving pair-wise, gapped global alignment problems. As with any dynamic programming algorithm, the optimal solution is calculated from the optimal subproblem solutions. A matrix of size query length $\times$ subject length is created. We call this matrix *H*. $H_{i,j}$ is the score of the best global alignment up to residue *i* in the query and residue *j* in the subject. Therefore, the dynamic programming recursion of Needleman-Wunsch, assuming a linear gap penalty model, can be written as:

$$H_{i,j} = \max\{H_{i-1,j-1} + S_{i,j}, H_{i-1,j} - d, H_{i,j-1} - d\}$$

In this equation, $S_{i,j}$, which is derived from the score matrix, is the score of aligning residue *i* of the query with residue *j* of the subject sequence and *d* is the score of inserting a gap in either the query or the subject. The first term in the above formula corresponds to a case where residue *i* is aligned to residue *j*, whereas the second and third terms correspond to a cases in which a gap is inserted to either the query or the subject. The recursion is initialized by $H_{0,j} = -j \times d, H_{i,0} = -j \times d$ to account for the initial gaps. The best alignment score is calculated at the bottom right corner of the matrix, and the actual alignment can be written by tracing back the dynamic programming matrix calculation from the bottom right corner.

In order to calculate the affine gap penalty, the recursion should be rewritten as follows:

$$E_{i,j} = \max\{H_{i,j-1} - u_d, E_{i,j-1} - v_d\}$$
$$F_{i,j} = \max\{H_{i-1,j} - u_q, F_{i-1,j} - v_q\}$$
$$H_{i,j} = \max\{H_{i-1,j-1} + S_{i,j}, E_{i,j}, F_{i,j}\}$$
where
$$H_{0,0} = F_{0,0} = E_{0,0} = 0,$$
$$E_{0,j} = H_{0,j} = -u_d - (j-1) \times v_d \quad \textit{and} \quad F_{0,j} = 0 \ \textit{for} \ j > 0,$$
$$F_{i,0} = H_{i,0} = -u_q - (i-1) \times v_q \quad \textit{and} \quad E_{i,0} = 0 \ \textit{for} \ i > 0$$

In the equations above, $E_{i,j}$ calculates the penalty of inserting a gap into the subject sequence in the affine gap penalty model, $F_{i,j}$ calculates the corresponding value for the query sequence, and $H_{i,j}$ is the score of the best global alignment up to residue *i* in the query and residue *j* in the subject.

For a query of length *q* and a database sequence of length *d*, the running time of this algorithm is $d \times q$. This algorithm is guaranteed to find an optimal solution, but its running time makes it impractical for large database searches.

Perhaps the most renowned dynamic programming algorithm in biosequence analysis is the Smith-Waterman algorithm [Smi81]. This algorithm is used to find the optimal gapped local alignment between a query and a database sequence. The algorithm is very similar to Needleman-Wunsch, and its recursion for linear gap penalty is as follows:

$$H_{i,j} = \max\{H_{i-1,j-1} + S_{i,j}, H_{i-1,j} - d, H_{i,j-1} - d, 0\}$$

The initialization condition for this recursion is $H_{i,0} = H_{0,i} = 0 \quad \textit{for} \ i \geq 0$.

Similar to Needleman Wunsch, in case of an affine gap penalty system, the recursions are rewritten as follows:

$$E_{i,j} = \max\{H_{i,j-1} - u_d, E_{i,j-1} - v_d\}$$
$$F_{i,j} = \max\{H_{i-1,j} - u_q, F_{i-1,j} - v_q\}$$
$$H_{i,j} = \max\{H_{i-1,j-1} + S_{i,j}, E_{i,j}, F_{i,j}, 0\}$$
where
$$E_{0,j} = H_{0,j} = F_{0,j} = 0 \text{ for } j \geq 0,$$
$$F_{i,0} = H_{i,0} = E_{i,0} = 0 \text{ for } i \geq 0$$

As can be seen, the only difference between this recursion and Needleman-Wunsch is the addition of zero in the best subsequence alignment score calculation. This small change enables us to align a portion of the database and sequence and find the best local alignment. The best alignment can be generated by tracing back from the element with maximum score in the *H* matrix until we reach an element with a score of zero. Similar to the previous algorithm, for a query of length *q* and a database sequence of length *d*, the running time of Smith-Waterman is $d \times q$. The algorithm is guaranteed to find the best local alignment with possible gaps, but its slow running time in comparison with heuristic methods makes it less practical for large database searches.

3.6 BLAST

3.6.1 Overview

BLAST is the most dominant heuristic approximate string-matching tool for finding either gapped or ungapped local alignments between a query and a large collection of database sequences [Alt90]. Although Smith-Waterman is guaranteed to give optimal results, there are two major reasons why BLAST is the standard approximate matching search tool for proteins and DNA. The first reason is speed: NCBI BLAST can be 50 to

100× faster than Smith-Waterman, and it almost always provides the better quality results. The speedup gain of BLAST is due to the heuristics that it uses to find the best alignments. The second reason for the dominance of blast is that it avoids junk matches thanks to complex heuristics and statistics. Using a naïve Smith-Waterman algorithm will cause several meaningless alignments to be reported.

The fundamental idea of BLAST is to avoid searching the entire database by finding hot spots in the database sequences that can potentially result in high-scoring alignments. To this end, NCBI BLAST is divided into three stages, namely word matching, ungapped extension, and gapped extension. We will look at these stages in the following sections. This overview is based on that in [Kor03].

3.6.2 Word Matching

The first step is to find short stretches of high similarity between the query and the subject sequence. Here, a *w-mer* represents a substring of length *w* on either the subject or the query sequence. The first algorithm, which is called the single hit algorithm, finds the identical w-mers between the query and the subject for DNA and matches with high scores for proteins. This approach is typically used in DNA searches with a default length of 11. The matches are called seeds.

Another algorithm, often used in protein sequence alignment, finds two matches of shorter length between the query and the subject sequence that are positioned close to each other and on the same diagonal. Here, the matching is not exact, and a threshold is used to find approximate matches of length *w*. More precisely, a match can be represented as a pair (d_0, q_0) where d_0 and q_0 are the coordinates of matching w-mers

on the subject and the query, respectively. let, (d_1, q_1) represent another match. A seed is detected when

$$0 < |d_0 - d_1| < A$$
$$d_0 - q_0 = d_1 - q_1$$

where A is a constant with default value equal to 40. Either the single hit or the two-hit algorithm gives us the coordinate of the seeds that can be extended by the ungapped extension phase.

3.6.3 Ungapped Extension

The second stage receives the seeds from the first stage and extends them to find high-scoring local ungapped alignments called high-scoring segment pairs (HSPs). The extension is performed to both the left and right of the seed. An early-termination mechanism is used: i.e., for each extension, a running score is maintained. Starting from the score of the seed, if aligning the next letters from the query and the subject increases the running score above the best value seen, then the alignment is extended to include the letters; if adding the letters reduces the running score by more than a constant X below the best running score seen during the extension, then the extension stops. If neither happens, the extension is continued, and the alignment is not enlarged. Once the extension is stopped, if the score of the extension is above a cutoff value, then the HSP is saved for the next stage, Otherwise, it is discarded.

3.6.4 Gapped Extension

The final step involves converting the ungapped HSPs from the previous stage into gapped alignments by extending them to the left and right and adding gaps if necessary. This stage also uses an early-termination algorithm to minimize the

extension time. The gapped extension uses DP-like mechanisms similar to that in Smith-Waterman.

3.6.5 Statistical Evaluation in BLAST

NCBI BLAST reports scores and E-values as measures of the significance of alignments. For a given query and subject pair, the reported E-value shows the expected number of alignments with the resulting score. The smaller the E-value, the more significant the alignment, meaning that there is a smaller chance of having such an alignment by random noise.

For a query of length q, a database of length d, a score of S between the query, and a subject sequence from the database, the E-value is determined by

$$E - Value = kq'd'e^{-\lambda \times S}$$

where k and λ are Karlin-Altschul constants calculated from previous simulations [Kor03]. q' and d' are the effective lengths of the query and the database, respectively. The idea of effective length for a query and database comes from the fact that an optimal alignment usually starts far from the right edge of the sequence [Kor03]. Here, q and d represent the length of the query and the database, respectively. Additionally, let N represent the number of subject sequences in the database. The effective lengths of the query and database can be calculated by the following formulas:

$$q' = q - l$$
$$d' = d - N \times l$$

Here, l is an integer value that is called length adjustment, which is calculated by the BLAST program. Another important parameter in the BLAST statistic is called effective

search space, and it is equal to $q' \times d'$. Any alteration of these parameters will result in incorrect statistical results being reported by the program.

3.7 CLUSTAL-W: Multiple-Sequence Alignment

Another task that biologists routinely perform is the extension of pair-wise sequence alignment to multiple-sequence alignment (MSA). MSA is used to find the evolutionary relationship between sequences, to find homologous regions in a groups of sequences, or to conduct phylogenetic analysis. For sequences that are not closely related, finding an accurate MSA is a topic of extensive research. MSA can be an expensive algorithm, both in the time and the amount of space required. Accelerating MSA alignment algorithms not only provides a better means for biologists to perform their routine tasks, but it also can assist them in finding better alignments, which can result in improvements in the accuracy of the MSA. In the following sections, we will look at some of the well-known algorithms for performing MSA.

3.7.1 Dynamic Programming

We can extend the dynamic programming recursion of the pair-wise sequence alignment to multiple sequences. In this case, a multidimensional dynamic programming solution is used. This approach quickly becomes intractable as the number of sequences grow. Thus, it is worthwhile to notice that using dynamic programming as a solution for MSA is only used for very small sets of sequences.

3.7.2 Progressive Multiple-Sequence Alignment: ClustalW

The most commonly used method in MSA is the progressive sequence alignment method, which was originally introduced by Fong and Doolittle in [Fen87]. There are

several similar progressive MSA methods which vary in some accuracy and performance details [Tho94], [Not00], [Edg04],[Hig98],[Kat02]. In general, the skeleton of progressive sequence alignment algorithms consists of the following three stages:

1) The first stage of the algorithm is to construct a distance matrix. For each pair of sequences, a pair-wise sequence alignment is performed, and some measurement of distance between the two sequences is stored in a matrix.
2) At stage two of the algorithm, a guided tree is generated using the distance matrix from stage one. This guided tree is generated using a clustering algorithm, such as neighborhood-joining or UPGMA.
3) At stage three, the final MSA is generated by following the order of the guided tree. Starting from the most similar sequences and moving in decreasing similarity, at each stage, two child nodes (which can be two sequences or alignments or profiles) are selected and aligned.

Different progressive alignment tools differ by the algorithms that they use in the three stages above and the subsequent optional optimizations they use to increase accuracy[Not00][Hig98][Edg04][Kat02].

ClustalW is one of the most widely-used progressive sequence alignment tools [Tho94]. In the first stage, for the construction of distance matrix, this tool uses a percentage of identities in the best local gapped alignment as a metric. In the second stage, it uses the classical neighborhood-joining classification to generate the guided tree. Finally, for the third stage, the tool performs a profile alignment. In a profile alignment, a group of sequences can be aligned with another group of sequences. In order to get the score of a position in this group-to-group alignment, the average of the all-to-all scores is used.

4 Previous Attempts to Accelerate Sequence Analysis

4.1 Overview

NCBI maintains a large database of biosequences. The increasing power of technology, advances in sequencing methods, and widespread interest in biosequence analysis have resulted in exponential growth in the size of this database. In addition, any database like the NCBI database should be able to respond to queries from all around the world in a timely manner. Obviously, the acceleration of tools that search, maintain, or analyze this vast and ever-growing database would be hugely beneficial. Also, as previously described, the increasing complexity of these tools provides additional motivation.

As explained before, there are several methods that can be used to query the sequence databases, but only a few of them are accepted as standard tools. Among these are the NCBI basic local alignment search tool (BLAST) and the Smith-Waterman algorithm. Smith-Waterman is substantially slower. As a result, NCBI uses the heuristic BLAST to query the database, and, as such, the majority of the bioinformatics community uses this tool. Any attempt to accelerate NCBI BLAST that results in a disagreement with the original version will not be accepted in the scientific community, even if the results have similar or even higher accuracy. On the other hand, even though the acceleration of NCBI BLAST is important, the software package is highly optimized and complex: many levels of optimization have been added since the original algorithm was proposed. This poses great challenges to any attempt to accelerate it.

As we will see, there have been many attempts to accelerate NCBI BLAST using traditional cluster computing methods. However, these systems usually incur excessive power consumption and high costs. An FPGA-based accelerator can deliver the same performance with significantly less power consumption and fraction of the nodes.

Several academic and industrial attempts have been made to accelerate sequence analysis algorithms. These works include pure software optimizations on shared multiprocessor systems, FPGA or ASIC-based hardware accelerators, GPU based systems, cloud computing and clusters of computers.

When comparing a query to database sequences, the query can be compared with each subject sequence independently of other subject sequences. With this observation, we can notice that comparing a query to a database of sequences is an embarrassingly parallel problem. This is the basis for the so called database segmentation approach for the acceleration of sequence analysis [Dar03]. In the database segmentation approach, the database is divided into smaller portions, and each portion is assigned to a processing unit. This is referred as inter-task parallelization.

On the other hand, to further increase the speed of a system, especially when using an accelerator, one needs to parallelize at a finer granularity. This level of parallelism is called intra-task parallelism, and it refers to parallelism inherent in the comparisons of the subject characters against query characters.

Usually, the techniques used in software optimization of sequence analysis tools are based on either cache efficiency considerations or reducing the number of required instructions in kernel portions of the code. These techniques are hardly useful for hardware implementations. Most of the time, the dynamic programming recursions are implemented with a systolic array in hardware. However, the recursions of these applications are so computationally intensive that the operating frequency of hardware

are below expectations. Pipelining these recursions is not necessarily useful, either. Similar problems to what we have mentioned above occur with hardware-accelerating heuristic applications such as NCBI BLAST. This makes direct mapping of these software-minded applications to hardware a dubious choice.

In this chapter, we will investigate the previous approaches to accelerating sequence-analysis tools. The rest of this chapter is as follows. In Section 4.2, we will look at the best implementations of the Smith-Waterman algorithm. Section 4.3 reviews the previous attempts for hardware acceleration of the Smith-Waterman algorithm. In Sections 4.4 and 4.6, we will review the previous cluster-based and accelerator-based attempts to accelerate NCBI BLAST. In Section 4.7, we will investigate previous attempts to accelerate multiple sequence alignment applications.

4.2 Software Acceleration of Smith-Waterman

Attempts to accelerate Smith-Waterman date back to the mid ‘90s. One of the first attempts to map Smith-Waterman to an SIMD architecture is reported in [Alp95]. Alpern et al. used a combination of optimizations towards a cache-efficient code and SIMD-based parallelism and achieved a modest speedup over a very early implantation on an i86 processor.

Wozniak et al. presented an implementation of Smith-Waterman on a Sun Ultra Spark processor using its SIMD video instructions [Woz97]. Their work is based on the intra-task approach, and it uses the SIMD instructions to parallelize a Smith-Waterman tableau's cell updates. The key observation is that the cells along the antidiagonals of the alignment tableau can be processed independently. An example of an antidiagonal

is shown in Figure 4-1. Using this approach, Wozniak et al. achieved 2× speedup over the best serial code of their era.

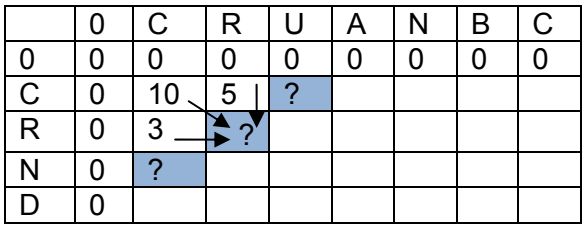


Figure 4-1 Inherent Parallelism in Smith Waterman Antidiagonals

T. Rognes and E. Seeberg were the first to use SSE/MMX instructions set to implement an SIMD version of Smith-Waterman on an Intel processor [Rog00]. They also introduced the concept of the query profile. Using a query profile reduces the number of score table lookups in the inner loop of the Smith-Waterman recursive implementation. Thanks to SIMD implementation, query profiling, efficient usage of cache, and some other optimization techniques they achieved 6-fold speedup over a highly optimized serial Smith-Waterman.

Farrar used SSE2 to implement a SIMD version of Smith-Waterman [Far07]. In contrast to previous work, Farrar's query profile is stripped so that the access pattern to the query profile is more efficient. Because of this improved access pattern, fewer instruction are executed in the inner loop of the Smith-Waterman dynamic programming C code. Farrar also proposed using a lazy F function, which helps to minimize the conditional branches inside the inner loop. As a result of these optimizations, the code achieved a 2× improvement over previous SIMD implementations.

Another Smith-Waterman implementation, which is called SWPS3, is an integration of SIMD and a multithreaded implementation [Sza08]. It can be mapped to both IBM Cell or to x86/SSE2. The code is based on Farrar's intra-sequence parallelization. It extends Farrar's implementation to IBM Cell, implementing a multithreaded version.

A similar implementation of Smith-Waterman on PS3 (called CBESW) is described in [Wir08]. It is an inter-sequence SIMD implementation of Smith-Waterman, and it achieves up to 3.4 GCUPS (Giga Cells Updates Per Second).

A faster implementation of Smith-Waterman was introduced in 2011 by Rognes [Rog11]. The implementation is available for the general public under the name SWIPE. The idea was to use SSSE3 instructions to implement an inter-sequence parallelization of Smith-Waterman. Each subject sequence is mapped to a portion of SSE instruction. Using six cores, a multithreaded implementation of the code achieves 106 GCUPS.

There are a number of attempts to map the Smith-Waterman algorithm to GPU [Lip88]. CUDASW++2.0 implements Farrar's stripped query profile-based implementation on GPUs. It utilizes both inter- and intra-sequence parallelism and achieves an average of 16.5 GCUPS.

4.3 Hardware Acceleration of Smith-Waterman

The first attempts to accelerate Smith-Waterman using special-purpose hardware were done in the late 1980s. P-NAC is considered the first hardware implementation of Smith-Waterman [Lop87]. It computes the edit distance between genome sequences.

Since the introduction of P-NAC, we have seen many improvements in the implementation of genomic Smith-Waterman on hardware. However, these optimizations do not apply for proteomic Smith-Waterman. The reason behind this difference can be traced back to the differences in the scoring mechanisms used for genomic and proteomic sequences. For genomic sequences, only an edit distance is calculated. On the other hand, for proteomic sequences, a complete scoring matrix is needed.

One of the most well-known examples of these type of optimizations was presented by Lipton and Lopresti . They noticed that, if the gap penalty is set to one and the mismatch penalty is set to two, then the recursion can be rewritten as follows:

$$\mathrm{H(i,j)} = \begin{Bmatrix} \mathrm{H(i-1,j-1)\ if\ \Big(\big(H(i-1,j) or\ H(i,j-1)\big) = (H(i-1,j-1)-1)\Big)\ or\ (S=Q)} \\ \mathrm{H(i-1,j-1)+2} \end{Bmatrix}$$

Using this optimization, it is has been shown that the computation of an *H* matrix can be done in modulo-4 encoding [Lip87]. As a result, to record *H*, only 2 bits are required in each cell in the alignment tableau. It is clear that such optimization is not practical when more complicated scoring mechanisms are used. For example, one cannot use this optimization for proteomic sequence alignment. As a result, hardware implementations of genomic Smith-Waterman are an order of magnitude faster than hardware implementations of proteomic Smith-Waterman.

The first hardware implementation of Smith-Waterman that was capable of supporting protein score tables and affine gap penalties was introduced in 1991 by M. Waterman [Cho91]. This work, which was called BISP, was the basis of all the future

improvements in hardware implementation of proteomic Smith-Waterman. The BISP architecture is based on data-flow graph analysis of the Smith-Waterman algorithm. The following Figure shows the Smith-Waterman algorithm's dependence graph.

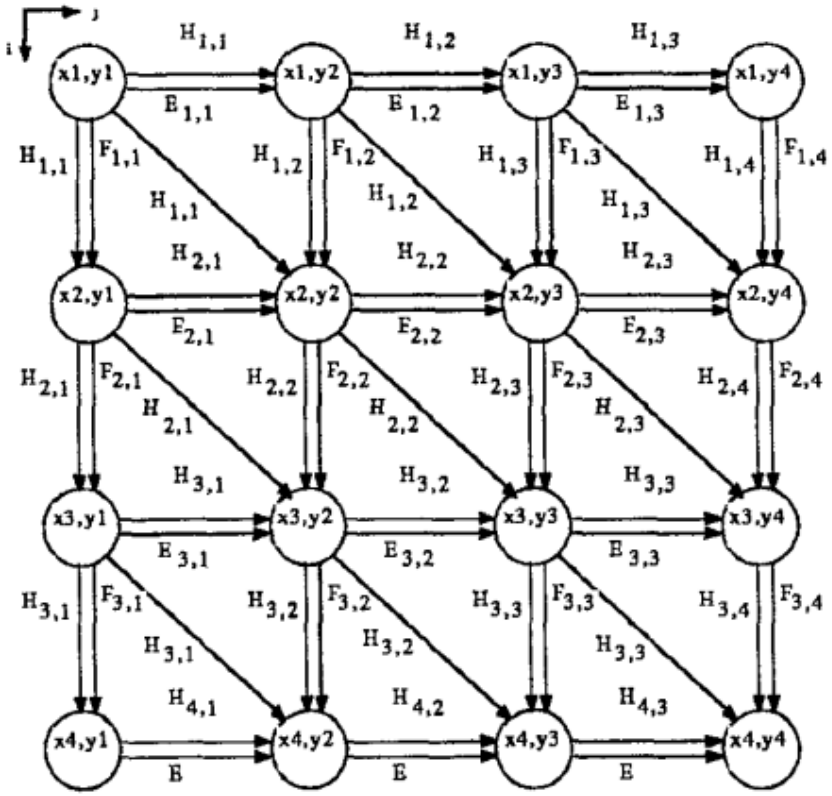


Figure 4-2 BISP Data flow Graph [Cho91]

As shown in Figure 4-2, from the analysis of a data dependence graph (DG), a signal flow graph (SFG) is derived. The signal flow graph assigns the virtual nodes of the DG to the actual processing elements in the SFG. The hardware implementation of Smith-Waterman is a systolic array consisting of identical processing elements chained together. In this systolic array, there is a one-to-one mapping between processing elements of the systolic array and nodes of the SFG. With careful analysis of the SFG,

the structure of each processing element is derived. The structure of each processing element is shown in Figure 4-3.

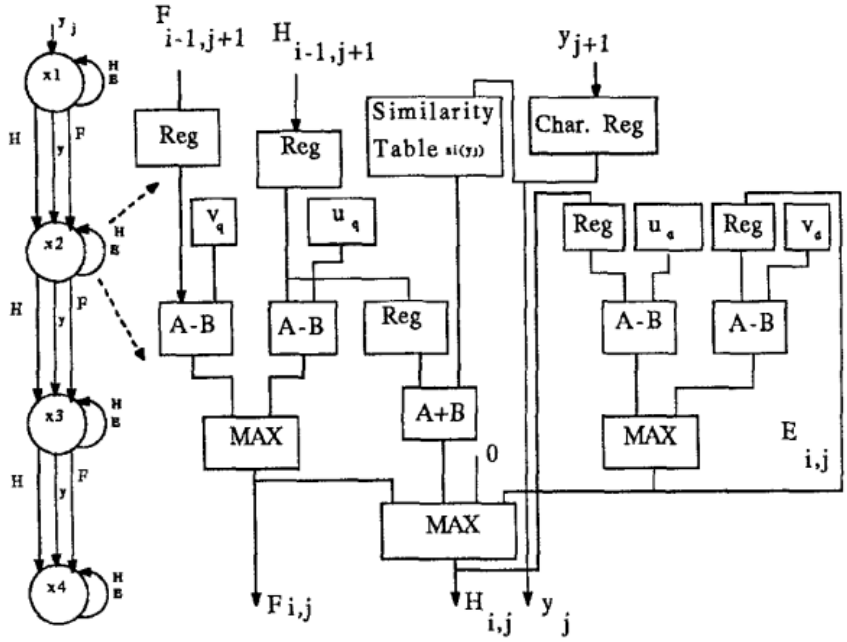


Figure 4-3 BISP Processing Element

Because FPGA resources are limited, one needs to fold the pipeline of the systolic array in order to support large queries. Oliver et al. proposed a method to do so that was based on the processing element described in BISP [Oli05] . If a query length is larger than the maximum number of processing elements (PEs) available on the target FPGA, the query is divided into multiple portions. The entire subject sequence is streamed through the systolic array in multiple passes. At each pass, a portion of the alignment tableau is generated (see Figure 4-4). A FIFO is used to store the intermediate results corresponding to the last characters of each query segment. When processing the next query segment, the contents of the FIFO are streamed through the systolic array. The following figure shows the idea of a folded Smith-Waterman. Figure 4-5 shows the

architecture of Oliver's Smith-Waterman [Oli04]. Oliver's implementation achieved 5.8 GCUPS on a virtex II FPGA.

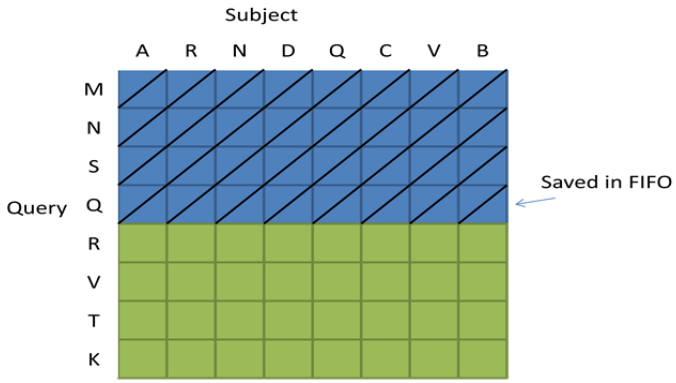


Figure 4-4 Folded Smith Waterman

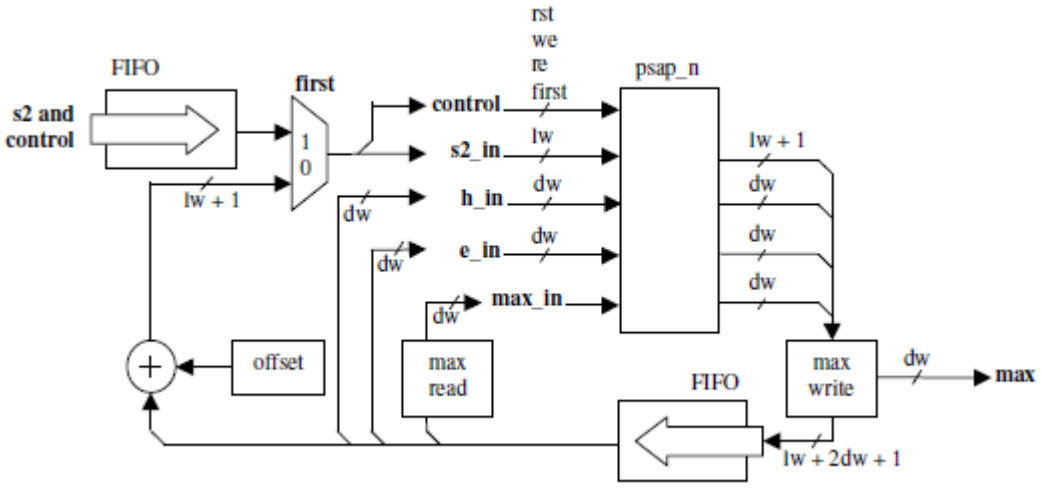


Figure 4-5 Oliver's SW Processing Element [Oli04]

The most challenging problem with hardware implementation of Smith-Waterman is the long critical path that limits the operating frequency. For most target FPGAs the critical path is inside the PE. In Figure 4-3, the critical path is on the feedback path from *E* to *H*

to *E*. Although this design can be further pipelined to increase the operating frequency, breaking the feedback path of the PE poses two challenges to the designer.

First, since the value of the each diagonal is dependent on the calculated values of the previous diagonal, the addition of any pipeline stage requires an additional idle clock cycle. In other words, the pipelined PE should always wait for the proper values of *H*, *E*, and *F* to be appear in its inputs so that it can start the calculation. Since the calculation of the feedback path of each PE does not overlap with other PE calculations, further pipelining requires additional processing cycles.

Ideally, in a systolic array, to process a subject sequence of length *d* against a query sequence of length *q*, *d*+*q* clock cycles are required. This brings us to the second problem in pipelining the PEs of Smith-Waterman: with finer grain pipelining of Smith-Waterman's PEs, we will need more clock cycles to fully process each subject sequence. In other words, if *d* and *q* are the lengths of a subject sequence and a query, respectively, and if each PE is pipelined *n* times, then the number of required clock cycles to process the subject sequence becomes *n(d+q)*. It is clear that simple pipelining of Smith-Waterman PEs will not improve the end-to-end performance.

Zhang et al. proposed a method to implement the "max" operations of the Smith-Waterman recursion with minimal area on Altera FPGAs [Zha07]. However, in order to optimally map their max operations to FPGA, they needed to add a flip flop at the end of each max function. As a result, their design required the same multistage processing mentioned for the fine-grain pipelining. In order to work around this issue, Zhang et al. implemented a multiphase Smith-Waterman processing element. In their

implementation, they used four different clocks with similar frequencies and different phases shifts. Using these clock signals, they clocked different registers of the PE with different clocks, thus reducing the effect of multistage pipelining. Zhang's implementation achieved 25.6 GCUPS on a stratix II FPGA.

4.4 Cluster Computing and NCBI BLAST

From a high-level point of view, one can approach the problem of parallelizing NCBI BLAST in two different ways. On one hand, the incoming queries can be distributed among multiple processing nodes. This way, one can increase the system throughput. This approach is called query segmentation or inter-query parallelization in literature. In this approach each one of the processing nodes works independent of the other ones. Ideally, in order to avoid memory stalls and routing contentions, each node should have its own local copy of the entire database. This is a drawback, considering the exponential growth rate of the genomic databases. This drawback is exacerbated by the fact that the current genomic databases don't fit entirely on a memory module and should be read from a hard disk. The second approach is to divide the database among multiple processing nodes. This is referred to as database segmentation or intra-query parallelization. Considering the parallel nature of NCBI BLAST, this approach seems reasonable. In order to accomplish this, there should be a mechanism to collect the results from a set of worker nodes and produce the final result in the required format. In [Dar03], the implications of database and query segmentation have been studied.

There have been several software-based attempts to parallelize NCBI BLAST. E.H. Chi. et al. studied the efficiency of shared memory multiprocessors for sequence similarity search problem and concluded that as long as the database fits in the memory of an individual computation node, and no memory access contention occurs between the processors, linear scalability in response time and throughput is achievable [Chi97]. This test cases included up to 24 processors.

R.C. Braun et al. [Bra01] used a job scheduler system to submit queries to different nodes of a cluster of workstations. They showed the possibility of using workstation clusters to increase the throughput of the blast services. TurboBLAST [Chi02] parallelizes BLAST on a cluster of workstations, supercomputers, or grids. It uses a java virtual machine to transparently parallelize BLAST. Each worker node works on a portion of a database, and a master node merges the results. Similarly, mpiBLAST parallelizes BLAST using an MPI interface with a database segmentation approach [Dar03]. There have been numerous other attempts to parallelize BLAST, and all have the same idea of database segmentation and query batching with minor differences in underlying job scheduling platforms, algorithms, and support for fault tolerance and database updates[Mat03][Gar06].

4.5 GPU accelerated NCBI BLASTp

There have been a number of attempts to accelerate BLASTp on GPU. Liu et. al. used GPUs to accelerate NCBI BLAST in CUDA-BLAST [Liu11]. They used a combination of coarse grain and fine grain parallelization techniques to map NCBI BLASTp alignments to GPU threads [Liu11]. Using a GeForce GTX 295, CUDA-BLAST achieves 3x to 4x

speedup over a quad core Intel CPU. A similar work that only uses coarse grain parallelization is reported in GPU-BLAST [Pan11] in which each GPU thread handles a separate subject sequence. In order to load balance the thread the database sequences are sorted based on their lengths. Using an NVIDIA Fermi C2050 GPU , the authors reported 1.5x speedup over 4 threaded NCBI BLAST. Another attempt to accelerate NCBI BLAST using GPUs [Lin10] reports 1.7x to 2.7x speedup over single threaded executions using an NVIDIA GeForce 8800 GTX GPU . In conclusion, generally a speedup of 3x to 4x over multithreaded NCBI BLAST appears to be achievable.

4.6 FPGA Accelerators and NCBI BLASTp

In this section, we will describe the most important attempts to accelerate NCBI BLASTp on FPGAs, namely Tree BLAST and Mercury System. There have been some other attempts to accelerate other versions of NCBI BLAST, such NCBI BLASTn for DNA databases or tBLASTx to search a protein sequence against a DNA database [Mur05][Eur07]. These early works mostly focused on DNA version of BLAST which is the simplest of all BLAST versions[Mur05]. We focus on the protein version of BLAST, BLASTp.

4.6.1 Tree BLAST

Tree BLAST was first introduced in [Her07] as an attempt to develop a compact and regular hardware structure that emulates the ungapped extension phase of NCBI BLAST. Tree BLAST consists of a set of processing nodes that are arranged in a binary tree structure, as shown in Figure 4-6. The query profile is loaded into the leaves of the tree. The subject sequence is streamed across the leaves of the tree, and one

complete score sequence is generated every cycle. Each score sequence corresponds to a global ungapped mapping of the subject and query characters. Each node processes two scores. The generated score sequences are processed by the tree to find the best local ungapped alignment at the root node. The operation of each node is as follows: each node of the tree maintains four integer variables, Max, Sum, LeftRunScore, and RightRunScore.

For leaf nodes:

```
Sum = Left+Right
LeftRunScore=Max(Left,Sum,0)
RightRunScore =Max(Right,Sum,0)
Max =Max(Sum,Left,Right,0)
```

For internal nodes:

```
Sum = Left.Sum+Right.Sum
LeftRunScore=Max(Left.LeftRunScore, Right.LeftRunScore+Left.Sum )
RightRunScore=Max(Right.RightRunScore, Left.RightRunScore+Right.Sum )
MaxScore =Max(Left.max,Right.max,Left.RightRunScore+Right.LeftRunScore)
Sum=Left.Sum+Right.Sum
```

It has been proven that, through the use of these nodes, the root node will output the score of the best local ungapped alignment between the two sequences [Her07]. The tree structure has several features that make it suitable for hardware implementation.

1. The tree structure can be pipelined as deeply as required.

2. The tree structure has a very compact construction that maps well into the hardware.
3. The tree structure can be structured to an arbitrary size with no additional complexity.
4. Folding the tree is easy to accomplish without additional hardware overhead. This makes it possible to trade area with speed.

Figure 4-6 Tree BLAST structure shows a Tree BLAST example.

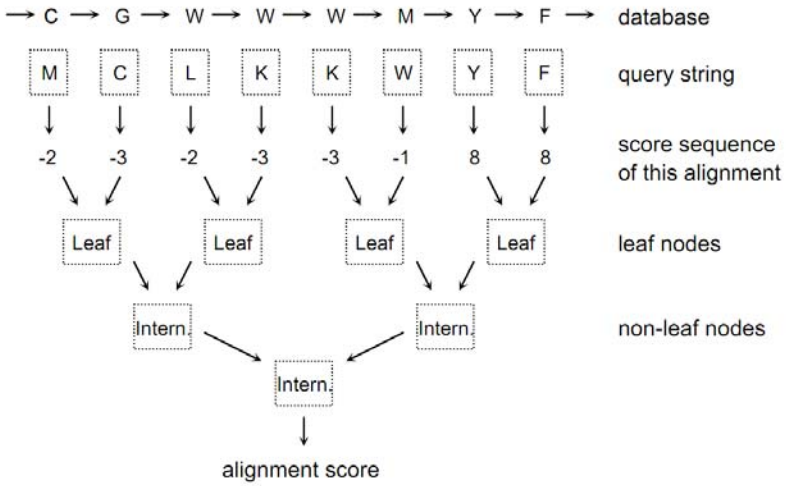


Figure 4-6 Tree BLAST structure

Tree BLAST supports the following fundamental options:

- **Folding:** The tree can be folded to support queries that are larger than what can fit on the chip. In this case, a portion of the tree is examined at each clock cycle. For example, if the tree is folded four times, ¼ of the query is mapped to the tree,

and, at each clock cycle, the score corresponding to ¼ of the query is generated. An update node at the root of the tree receives these four scores that correspond to different segments of the query, and generates the best score.

- **Replication:** Small trees can be replicated, thus allowing multiple queries to be processed simultaneously.
- **Arbitrary size:** Different tree sizes can be concatenated to generate trees with sizes that are not power of two. For example, Figure 4-7 shows how to generate a tree size of 1,664 characters from three tree binary trees.

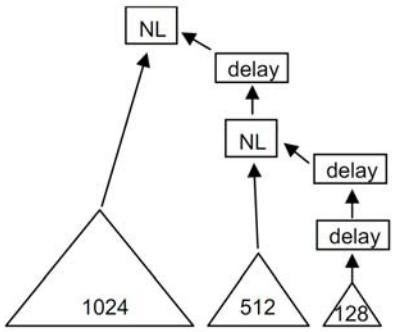


Figure 4-7 Arbitrary tree Size

Tree BLAST is used as the basic component in the initial version of CAAD-BLASTp [Par09]. CAAD-BLASTp implements a pre-filtering mechanism to accelerate NCBI BLAST. The basic design of CAAD-BLAST is to successively reduce the database (DB) without removing any potential matches. In the initial preprocessing stage, two thresholds are calculated: gapped and ungapped thresholds. These thresholds will be used by the filters in the subsequent stage. First, the DB is filtered by running Tree BLAST, and a reduced DB' is generated. Since all the alignments are examined, there is no need for the seed generation phase. As a result, the first phase of NCBI BLAST

can be skipped safely without jeopardizing the agreement with NCBI BLAST. The reduced database (DB') contains all the sequences that, when compared with the query, score above a threshold (ungapped threshold). Then, Smith-Waterman is run to generate a further reduced database DB". In order to do this the smith-waterman scores are compared with the gapped threshold. Finally, DB" is formatted and sent to NCBI BLAST along with the original parameters and query.

In order to have correct results, the internal thresholds that NCBI BLAST use should be determined, and the E-values in the final report should be computed correctly. Also, CAAD BLASTp should ensure that DB" (i) contains all the sequences that NCBI BLAST would return and (ii) is sufficiently reduced so that the overhead of formatting DB" does not overwhelm any potential performance gain.

Figure 4-8 shows an overview of the steps required in CAAD BLASTp. The ungapped filter begins with the FPGA, along with the query and database, to compute the ungapped alignment scores. For the most promising sequences, scores are returned to the host, which uses them to specify DB'. For the gapped option, a new threshold is computed and passed to the FPGA, where the contents of DB" are determined. Finally, the reduced database (either DB' or DB") is formatted to be processed by NCBI BLASTp.

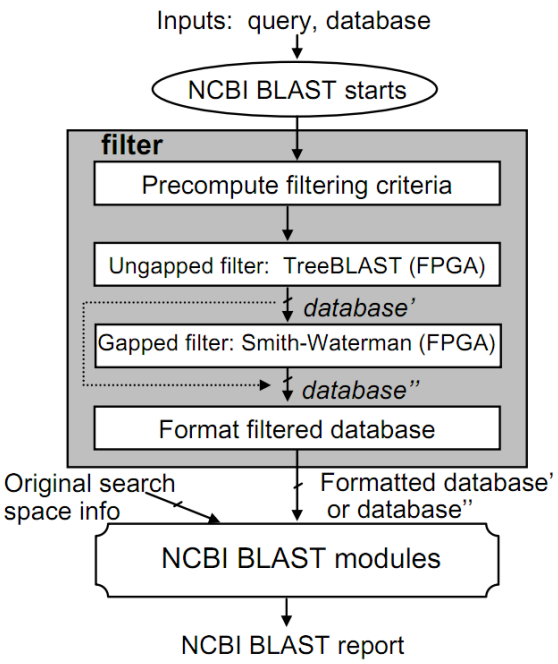


Figure 4-8 CAAD BLASTp Overview

Recall that NCBI BLAST returns a statistical significance report. NCBI BLAST code is integrated with the filters such that it reports the correct E-values and doesn't miss any sequences. In order to do this, the required parameters are calculated on the original database before the filtering process starts and are saved for the final stage. The profiteering mechanism seems efficient, but it has some drawbacks that can diminish performance.

One problem with CAAD BLASTp is that only one subject sequence can be processed at each time. As a result, when streaming the database, a number of null characters should be inserted between different subject sequences. The number of null characters should be equal to query length, a fact which can cause an average of 100% overhead.

The second major problem with CAAD BLASTp is that it ignores the seeding heuristic of NCBI BLAST. The seeding heuristic significantly reduces software runtime by limiting the positions in the database that need to be examined to a limited fraction of the entire database. CAAD BLASTp streams in the entire database and, thus, some performance gain is lost.

Third, running NCBI BLAST on a filtered database can be very time consuming because of large similarities between sequences and the query.

Fourth, the initial draft of CAAD BLASTp is not fully integrated with NCBI BLAST code. It requires reformatting the reduced database. This can potentially slow down the original binary.

CAAD BLASTp is integrated with the C toolkit of NCBI BLAST code. The C toolkit has since been replaced with a C++ toolkit. The C toolkit is slower and outdated, and the NCBI has stopped supporting its code.

Overall, these overheads can slow down NCBI BLAST runtime rather than speeding it up.

4.6.2 Mercury BLASTp

Contrary to CAAD BLASTp, mercury BLASTp implements the NCBI BLAST algorithm directly on FPGAs [Kri07]. Similar to NCBI BLAST, mercury BLASTp is a chain of three stages, as shown in the following figure.

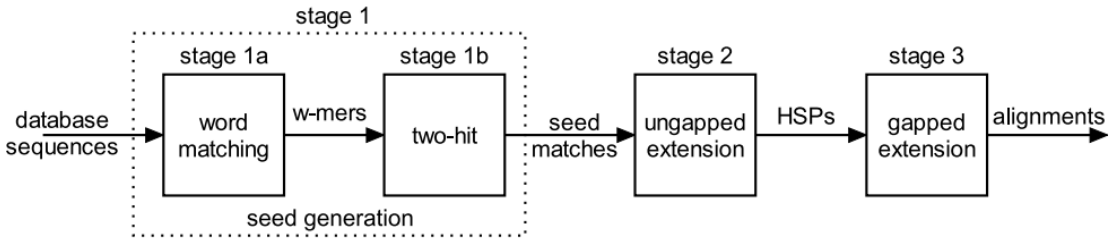


Figure 4-9 BLASTp Pipeline in Mercury[Kri07]

Profiling NCBI BLASTp shows that all three stages take a significant portion of its total runtime. As a result, the acceleration of all three stages is required to get a reasonable performance gain. A universally known way of implementing seeding in heuristic-based approaches of sequence analysis is the use of indexes (sometimes called profiles, or neighborhoods). NCBI BLAST creates an index of the query as well. A seeding index is a data structure that is used to find the seeds when comparing the query against a subject sequence. For a given word size and alphabet and for all possible combinations of words, the query neighborhood contains the indexes of all locations in the query that match above a threshold.

Similar to other types of NCBI BLAST, Mercury BLASTp uses an indexing approach to generate the seeds. In Mercury BLASTp, the query is indexed, and a query neighborhood is generated. The query neighborhood has all the information required to generate the seeds. For every possible w-mer, an entry in the lookup table stores a list of matching w-mer positions on the query (either an approximate or exact location that is based on the seed generation algorithm). As the database w-mers are scanned, these positions are retrieved from the lookup table and sent for further processing. In mercury BLASTp, the query index is divided into two parts: primary and secondary

tables. Each entry in the primary table contains up to three matching locations in the query. If the number of matching positions in the query is larger than three, the primary table stores the number of the matches and a pointer to the secondary table where the actual matches are stored consecutively. The query neighborhood is indexed for w-mers of length four, or 4-mers. As a result, it doesn't fit on the FPGA block RAMs and is stored off the chip on a memory module. The following figure shows the mercury system's lookup table data path and word-matching hardware design.

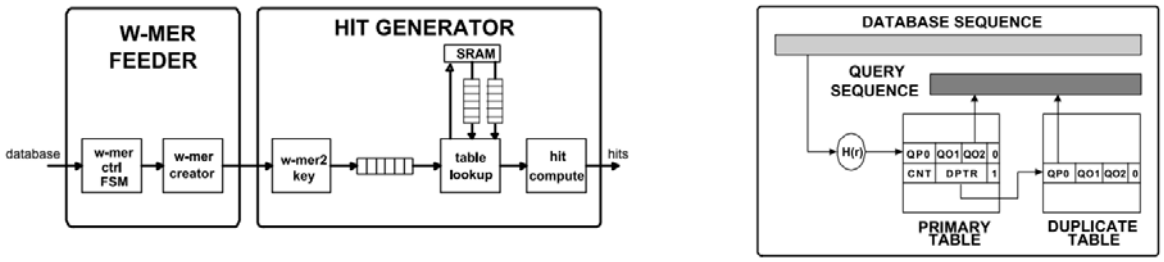


Figure 4-10 Mercury BLASTp Hit Generation

The generated hits are routed to two hit generation modules. In order to detect two hit seeds, an array is used which, for every diagonal, stores the position of the most recently encountered word match. Since sequence word matches can occur at any position in the subject in window of M diagonals where M is the query length, an array of length M should be sufficient, but the authors have used an array of length 2M.

The hits should arrive at the two hit units with the order that their database indexes indicate. Otherwise, there is a chance that some seeds might be missed.

In order to maximize the seed generation performance, both hit generation and two hit generation modules are replicated. Clearly, without replication, the seed generation can

become a bottleneck. Dedicated routing buffers steer data from the hit generators to the 2- hit units, as shown in the following figure.

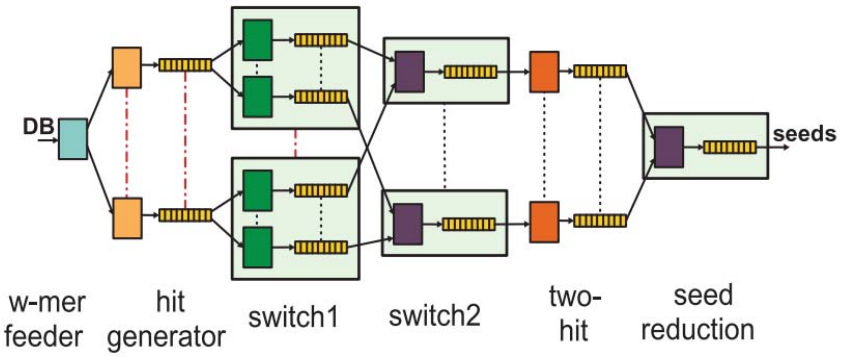


Figure 4-11 Two Hit Seed Generation

The replicated hit generation units access independent off-chip memory modules in parallel. In order to balance the workload of each hit unit, the diagonals are multiplexed amongst the hit units using the least significant portion of the diagonal numbers. The idea is that, because the hits occur in clusters close to each other, using low-order bits to assign diagonals to hit units can help in load balancing. Nevertheless, using multiple hit units and multiplexing the diagonals amongst them causes some challenging problems. Because the access time to the lookup tables depends on the number of matches, the seeds may not always arrive in their increasing database position. As a result, some seeds may not be detected. The authors have used a workaround heuristic that reduces the impact of this problem, and this heuristic results in reasonable accuracy (99%).

The next stage of the mercury system emulates NCBI BLAST'S ungapped extension. Recall that NCBI BLAST uses an early termination mechanism in the ungapped

extension phase. In extension with the early termination approach, seeds are extended in both directions. Extension in each direction is terminated as soon as it stops being promising; i.e., when the running score drops a certain threshold below the maximum score seen during the extension. This is done to reduce the workload of the CPU. Even though this optimization is suitable for software implementation, its hardware implementation is costly. As a result, mercury BLASTp uses another heuristic to simplify the problem. Instead of an early termination mechanism, mercury BLASTp examines a fixed window around each seed. The window size is 64 characters wide. With this approximation, mercury BLASTp implements ungapped extension with a dynamic programming algorithm. The design is mapped to FPGA as a systolic array. The resulting implementation achieves 96% to 99 % agreement with the reference.

Ungapped extension filters out most of the seeds. The promising seeds extensions generate a high-scoring segment pair (HSP) list which is passed to the gapped extension phase. Similar to the previous phase, mercury BLASTp performs the gapped extension with the fixed window approximation instead of the original early termination mechanism. This algorithm, which is basically a dynamic programming solution, is called banded Smith-Waterman, is mapped to a pipelined systolic array and is described in [Har07].

Mercury BLASTp is an efficient design. Using two Virtex II 6000 FPGAs, the authors have reached 10× to 15x speedups over CPU version. On the other hand, the biggest drawback is the approximations that have been used to simplify the hardware. Although the decrease in the accuracy seems insignificant, biologists tend to ignore any tool that deviates, even with smallest amount, from the standard NCBI BLAST. Therefore, having

a 100% accurate NCBI BLASTp acceleration that satisfies the cost effectiveness criteria is a challenging problem.

4.7 Acceleration of Multiple Sequence Alignment

In comparison to NCBI BLAST, there have been fewer attempts to accelerate multiple sequence alignment algorithms on hardware.

Oliver and et al. designed a systolic array to accelerate the first phase of Clustal-W [Oli05]. On the basis of the fact that the first phase of Clustal-W takes more than 90% of the overall runtime, they mapped the first stage of Clustal-W to FPGAs. Recall that the first stage of Clustal-W calculates a distance matrix, and the metric for the distance calculation is the number of identities in the best local gapped alignment between sequences. In order to count the number of identities in the best local ungapped alignment, Oliver et al. extended the dynamic programming recursive formula of the Smith-Waterman algorithm to count for the number of identities in the best local gapped alignment. Using this extension, they mapped the algorithm to a systolic array on FPGAs.

The idea of this extension is to count the identities based on the path taken in recursive relation of the Smith-Waterman algorithm. If Smith-Waterman aligns two characters, the identity condition is checked. Otherwise, the identity count is equal to the identity count in the direction of gap insertion.

For linear gap penalty, the extensions are as follows. The extension for affine gap penalty is similar. Given two sequences, S_1 and S_2, a substitution matrix (sbt) and a

linear gap penalty (α) and Smith-Waterman recursion relation (as described in section 3.5), the number of identities in the best local gapped alignment is given by $\mathrm{N(i,j)}$:

$$N(i,j) = \begin{cases} 0 & if\ H(i,j) = 0 \\ N(i-1,j-1) + m(i,j) & if\ \ H(i,j) = H(i-1,j-1) + sbt(S[i],S[j]) \\ N(i,j-1) & if\ H(i,j) = H(i,j-1) - \alpha \\ N(i-1,j) & if\ H(i,j) = H(i-1,j) - \alpha \end{cases}$$

$$where\ m(i,j) = \begin{cases} 1 & if\ S[i] = S[j] \\ 0 & otherwise \end{cases}$$

Using a VIRTEX II FPGA, Oliver et al. achieved a speedup of 50x in the first stage. Nevertheless, they did not implement the remaining stages on the FPGA, which, based on Amdahl's law, limits the end-to-end speedup to 10x .

Lioyd and Snell proposed a method to implement the third stage of Clustal-W on an FPGA [Lio11]. The third stage aligns sequences following the order of the guided tree. It performs a profile alignment for groups of sequences. The third stage takes almost the entire remaining 10% of the computation of MSA, and so it is critical to accelerate this stage in order to have reasonable speedup. Lloyd and Snell's profile alignment algorithm accelerated on an FPGA achieves 150x speedup over Clustal-W third stage.

There are a number of other works that accelerate Clustal-W on clusters of computers. ClustalW -MPI uses massage passing interface to parallelize Clustal-W on a cluster of workstations [Li03]. For the first stage, it uses a coarse-grain parallelism approach and achieves linear speedup. For the last stage, a combination of coarse-and fine-grain parallelism achieves 4.3x speedup using 16 processors.

There are also several attempts to accelerate Clustal-W on GPUs. MSA_CUDA reports the mapping of Clustal-W to a GeForce GTX 280 GPU [YSM09]. It uses both coarse- and fine-grain parallelism and maps all three stages to a GPU. A speedup to 37x is reported over a serial implementation on a Pentium 4 with results comparable to Clustal-W -MPI with 32 nodes.

5 CAAD BLAST

5.1 Overview

NCBI BLAST has three phases: identifying short sequences (words) with high-match scores (seeding), extending those matches without adding gaps (ungapped extension), and performing gapped extension on selected segments from the previous phase (gapped extension). For the sequences with the highest scoring alignments, an E-value (expected value) is computed from the raw alignment score and other parameters. Then, database sequences with sufficiently good E-values are reported.

Figure 5-1 shows a conceptual view of the three NCBI BLAST phases. In the first phase, hotspots in the alignment space of the subject and query sequence are found. The hotspots are those offsets of the query and subject sequences that satisfy the two-hit property, as described in section 3.6.2. The word size (w) is typically two or three for BLASTp, and the significance is determined on the basis of scoring performed with a scoring matrix, such as BLOSUM 62, and a threshold value. In the extension phase, seeds are extended in both directions to form high-scoring segment pairs (HSPs). Extension stops when it ceases to be promising; i.e., when the drop-off from the last maximum score exceeds a threshold of X. This is referred to as an early-termination mechanism. In gapped alignment, extension and evaluation are triggered only when an ungapped alignment satisfies the ungapped threshold. In this phase, seeds are extended in both directions to form real alignments, possibly by adding gaps to both sequences. Similar to ungapped extension, the early-termination mechanism is used; that is, extension stops when the dropoff from the last maximum score exceeds a

threshold of X. In gapped extension, the extension dropoff threshold X also depends on gap-opening and gap-extension costs.

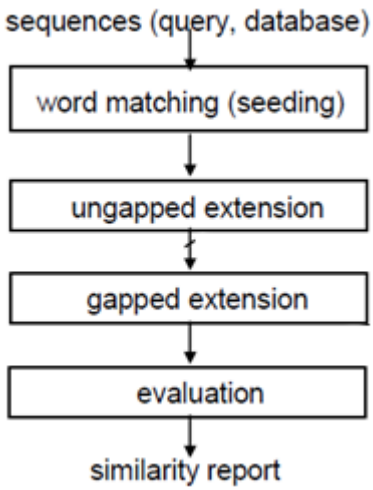


Figure 5-1 conceptual view of the three NCBI BLAST phases

The main idea in CAAD BLASTp is prefiltering; that is, to quickly reduce the size of the database to a small fraction and then use the original NCBI BLAST code to process the query. Agreement is achieved as follows. Prefiltering is guaranteed to be strictly more sensitive than the original code; that is, no matches are missed, but extra matches may be found. The latter can then be (optionally) removed by NCBI BLAST. The primary result is a transparent FPGA-accelerated NCBI BLASTp that achieves output identical to the original. Because the prefiltering mechanism is more sensitive than the original, the user may keep the extra outputs at no cost of performance.

The rest of this chapter is organized as follows: in Section 5.2, we describe a basic overview of the operation of the filters. Next, we describe the two-hit filter in detail. Section 5.4 describes the exhaustive ungapped alignment. Sections 5.5-5.7 have

details of the two main architectures of our CAAD BLAST system, results, and a scalability/portability study.

5.2 Filter Basics

CAAD BLAST uses three FPGA-based filters:

- The two-hit filter is based on the two-hit seeding algorithm. All alignments (all diagonals in Figure 1b) are evaluated based on whether or not they contain a two-hit seed. The output is a bit vector containing a 1 or 0 for each diagonal, depending on whether or not the diagonal contains a seed. We base our two-hit filter on the two-hit seeding algorithm used by Mercury BLAST and described in [Jac07].
- The exhaustive ungapped alignment (EUA) filter scores every possible alignment between the query and the database. For each sequence in the database, the filter returns the scores of the highest-scoring alignments. We base our EUA filter on the TreeBLAST algorithm described in [Her07].
- The exhaustive gapped alignment (S-W) filter is based on the Smith-Waterman algorithm and returns the highest-scoring gapped local alignments for each sequence in the database. We base our S-W filter on the version of the Smith-Waterman algorithm described in [Cho91].

Each filter reduces the amount of work that needs to be processed by the next filter. The two-hit pass provides “hints” to the EUA filter as to which diagonals can be skipped. As described below, actually skipping diagonals is not cost-effective, but making the EUA filters drastically more compact is. After compaction, the EUA pass is almost as

fast as the two-hit pass. The EUA filter prunes at least 95% of the database so that it does not need to be processed by the S-W filter. The S-W filter prunes the database to 0.1% of the original. The reduced database is then processed by NCBI BLASTp.

All three filters work on the same principal. Each occupies some amount of chip area (in the FPGA) and holds a copy of the query. Then, it executes as the database streams through it from off-chip memory. The filter size (in chip area) is related to the query size. Generally, the filter uses only a fraction of the chip area, and, therefore, it can be replicated a number of times. If the query is very large, then the filters will still operate correctly, but it will have reduced performance with a slowdown generally proportional to the query size. Thus, each filter thus runs in O(N), assuming that the query sequence is a small multiple of what can fit on a current FPGA, a characteristic of almost all proteins.

5.3 Two-Hit Filter

NCBI BLAST uses two-hit seeds to limit the number of diagonals that need to be examined. Only a small fraction of the entire stream size has the two-hit property and needs extension. These percentages as a function of query size are shown in Table 5-1.

Table 5-1. Fraction of alignments having two-hit property as a function of query size. Queries taken from the NR database.

Query size	average	Max
256	0.008	0.009
512	0.016	0.028
1024	0.02	0.025
2048	0.027	0.043

On the basis of the results in Table 5-1, we can see that restricting the stream to those hotspots that have a two-hit property can have a massive impact on the performance of a system. In order to exploit the two-hit heuristic in a hardware design, several issues need to be considered. First, ungapped extension is already very fast; it can be done in streaming rate (one residue per clock cycle) with the use of TreeBLAST. Thus, in order to improve, the two-hit filter should run much faster than one residue per clock cycle. Since this speed-up is unlikely to come from increased operating frequency, it must be possible to use the two-hit filter to extract more parallelism. This requires significantly greater potential replication of the generated two-hit cores than for the TreeBLAST which in turn requires that the two-hit unit cores be significantly smaller than the TreeBLAST cores. Second, the two-hit filters should improve TreeBLAST performance enough to compensate for the overhead they impose. The key idea in using a two-hit filter is to have multiple small filters that can work in parallel. Otherwise, there is no benefit in generating the seeds for a streaming design like TreeBLAST.

The two-hit filter is based on the NCBI BLASTp two-hit seeding algorithm. All ungapped alignments are evaluated as to whether or not they contain a two-hit seed. A bit vector is generated containing a 1 or 0 at each position depending on whether or not the corresponding alignment contains a seed. The basic function of a two-hit filter is shown in Figure 5-2. The design is generally similar to the one used in the Mercury BLAST seeding pass [Jac07].

Below, we will describe a single two-hit filter, and a description of an extension to multiple filters operating in parallel will follow immediately. We begin with some notes, an overview of the algorithm, and a critical observation.

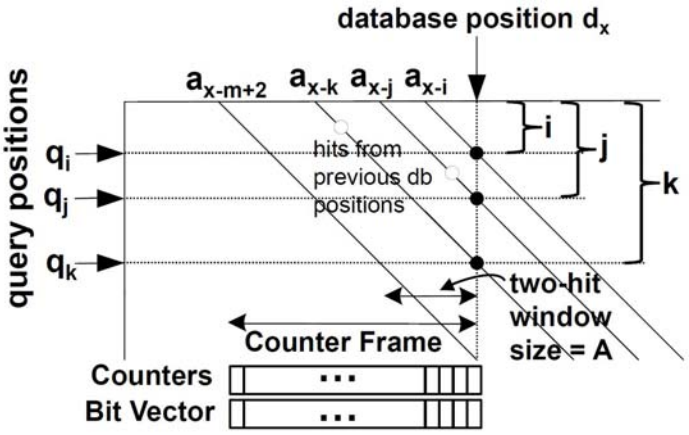


Figure 5-2 Two hit filter

Figure 5-2 shows the subject on the horizontal axis and the query on the vertical axis. Positions of each w-mer (a short sequence of w residues) are referred to as d_x for the database and q_y for the query. If the subject length is d and the query length is q, there are $d + q$ possible global ungapped alignments between the database and the query, which are represented by diagonals in the figure. We refer to each diagonal (alignment) as a_i. The output of the two- hit filter is a bit vector where each bit (b_i) corresponds to an alignment (a_i) and tells whether or not a_i has passed the filter (i.e., whether or not it has a two-hit seed). That is, a_i passes the filter if there are two hits within the distance threshold (A) (typically 40). If yes, then b_i is set; otherwise, it remains clear.

The basic data structure used to generate the two-hit seeds both in NCBI BLAST and in the Mercury BLAST system is a lookup table called position list or query index. For each possible w-mer, the position list stores the positions of all of the w-mers in the query that exceed the match threshold (typically 11) when aligned with that w-mer. The position list

has two parts: the primary and the secondary list. Recall that protein sequences consist of 20 characters. BLAST uses five additional special characters, making the alphabet size 25. NCBI BLAST's two-hit seeding uses w-mers of length 3 (3-mers) by default. Thus, there are 25^3 possible 3-mers. The primary list has an entry for each of the 25^3 possible 3-mers. For any possible 3-mer, if there are three or fewer hits in the query, then corresponding primary list entry holds all of those positions. If there are more than three occurrences, then the primary list entry contains the number of occurrences and the address in the secondary list where entries for those positions are written consecutively. A status bit indicates the record type.

For each alignment, we keep the position of the most recent hit, if there are any hits at all. When a new hit occurs on a diagonal, we compare its coordinates with the most recent hit on that diagonal to decide whether to issue a two-hit or not. Note that the hits on a given diagonal are generated in increasing order of their database position because the database is scanned from left to right. An overview of the operation of the two-hit filter is as follows. On iteration *x*, database 3-mer d_x indexes the position list. The query positions where matches occur, if any, are retrieved. Figure 5-2 shows three hits, at query positions q_i, q_j, and q_k. These correspond, respectively, to the *i*th position on alignment a_{x-i}, the *j*th position on alignment a_{x-j}, and the *k*th position on alignment a_{x-k}. This hit information is then used to determine whether another hit has occurred on any of these diagonals within the previous 40 positions (as shown in Figure 5-2 for alignment a_{x-j}).

The goal is to process the database at a streaming rate. Upon each iteration, a database 3-mer is processed. The advancement to the next iteration is made as soon as the hits corresponding to the current iteration are fetched from the position list.

The method is to use a frame of counters, one for each alignment where there could be match on the current iteration. Given that, on any iteration, only the last *q* alignments can be affected, the frame length is equal to this value. This is important because it makes hardware implementation feasible. As an example, for a hit in alignment a_{x-j}, the corresponding counter that is dedicated for that alignment is read, compared with *j*, and updated. If the difference between *j* and the previous value of the counter is less than *A*, then this indicates a two-hit hit occurrence for alignment a_{x-j} and bit b_{x-j} in the bit vector is set. If the distance between *j* and the previous value of the counter is more than *A*, the counter will update its last seen hit position to *j*. The counters for a_{x-k} and a_{x-l} are also processed similarly. However, for each alignment, advancement is monotonic; i.e., a hit on a later iteration will never be further back on the diagonal than the previous one. This guarantees the detection of all of the two-hits and a 100% agreement with NCBI BLAST. The overall architecture of the two-hit filter is given in Figure 5-3.

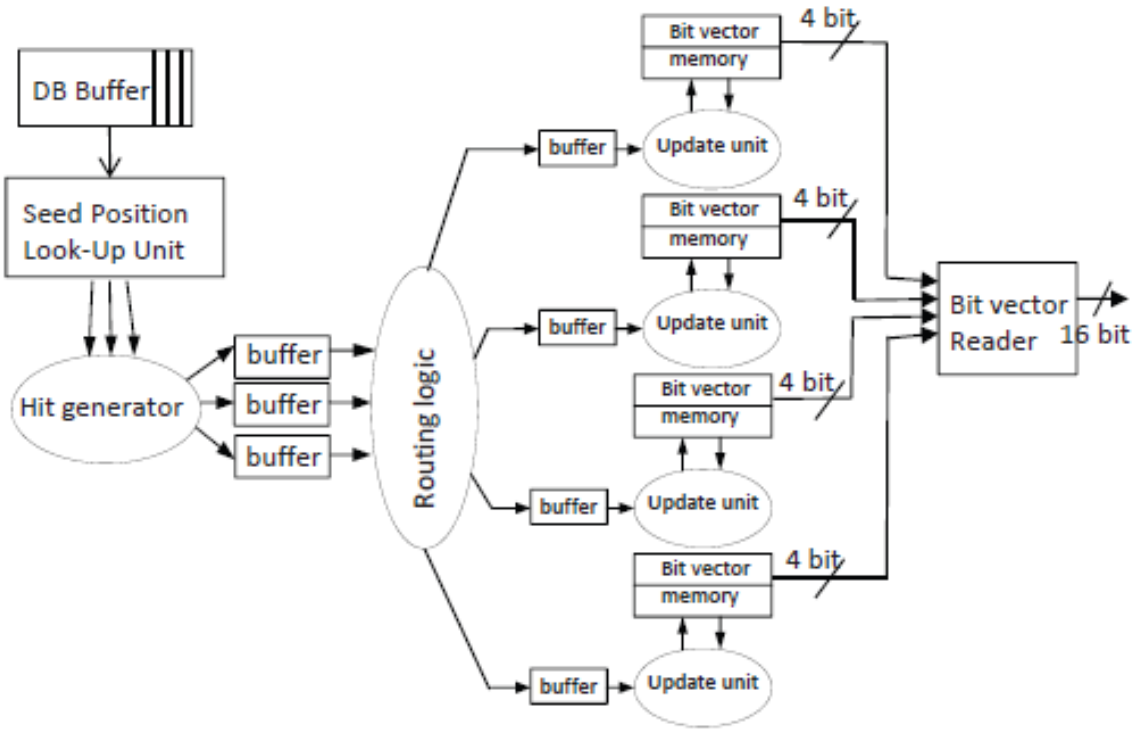


Figure 5-3 Two-Hit Filter Block Diagram

The hardware implementation consists of four major stages: the hit generation, the routing, the two-hit generation, and the bit-vector output. These four stages are described below.

I. The Hit Generation

The hit generator performs the following functions consecutively:

1. It reads the next character from the input database stream and increments the database position counter (subject index).
2. It forms a 3-mer and indexes the position list's primary section

3. It reads the data from poison list primary section and outputs the hits if any, if required indexes the secondary section of the position list and outputs the hits from the secondary section.

In order to process the database in streaming rate, the two-hit filter processes three hits at each clock cycle. Both the primary and the secondary list are structured to enable the fetch of three query positions per clock cycle. The following figure shows the general format of the position list entries. The first bit in the entry, called the status bit, is used to differentiate between two types of entries. If this bit is not set, then the entry contains up to three query positions. If the total number of positions is less than three, then the unused bits are filled with a special null data (i.e., -1). If the status bit is set, the entry holds a pointer into the secondary section, and the count of the matching words is stored consecutively in the secondary table.

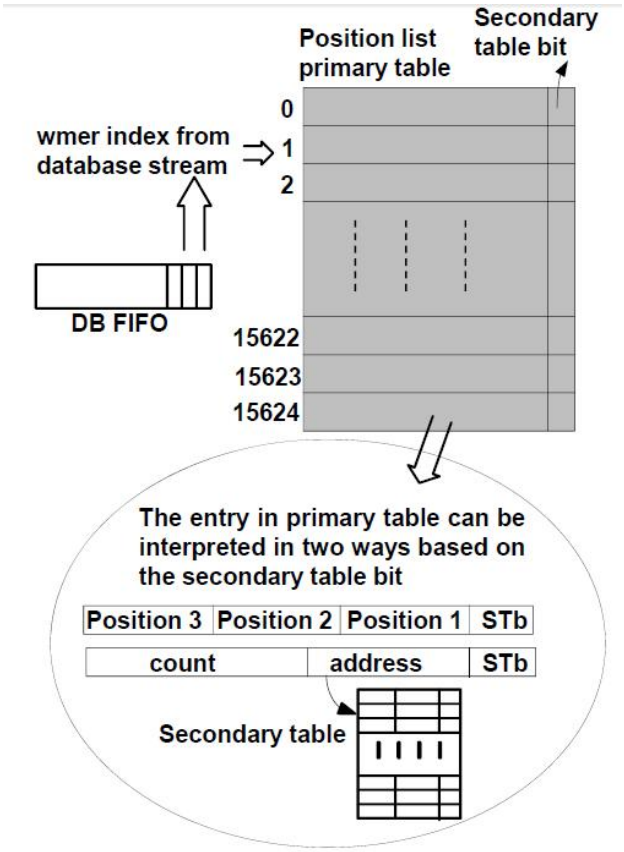


Figure 5-4 Two hit filter Query Neighborhood data structure

The first major task of the hit generator is to generate the addresses to the position list in streaming rate. There are three possibilities for the addresses:

1. It can be generated after a new character is read from the database and a new 3-mer is formed.
2. It can be the address in the pointer section of an entry retrieved from the primary section of the position list when the status bit is set to one.
3. It can be the previous address incremented by one while reading the secondary list.

Recall that each hit can be represented as a pair (d_x, q_y) where d_x and q_y are the coordinates of the matching w-mers in the database and query, respectively. Each hit generator has an internal counter that holds the index of the subject 3-mer that is being processed. At each clock cycle, if the status bit of the data from the position list is zero, then the three hit positions from the position list are paired with the database counter to generate the hit pairs. Otherwise, data is read from the secondary table, and up to three hits are generated per clock cycle in the subsequent cycles. The hits are written to the routing unit's input FIFOs.

II. Routing

The routing stage is responsible for routing the hits from the hit generator to the two-hit subunits. It has three input FIFOs that are written by the hit generator and four output FIFOs that are read by the four two-hit subunits. The hits are multiplexed to the four output buffers on the basis of the alignment they correspond to. The three matches are broadcasted to the four output ports. A hit with coordinates (d_x, q_y) is written to the output FIFO number $(d_x - q_y)\,\%\,4$. The output port's control logic checks the input matches to see if any of them should pass through that port. Each output port has an arbiter which selects inputs from multiple matches that might be required to be written to its output FIFOs. Higher priority is given to the matches with minimum subject position. In this way, the hits on a diagonal arrive in increasing database index to the two-hit unit, which is required if we want to detect all possible two-hits. The following figure shows a schematic of the routing circuit.

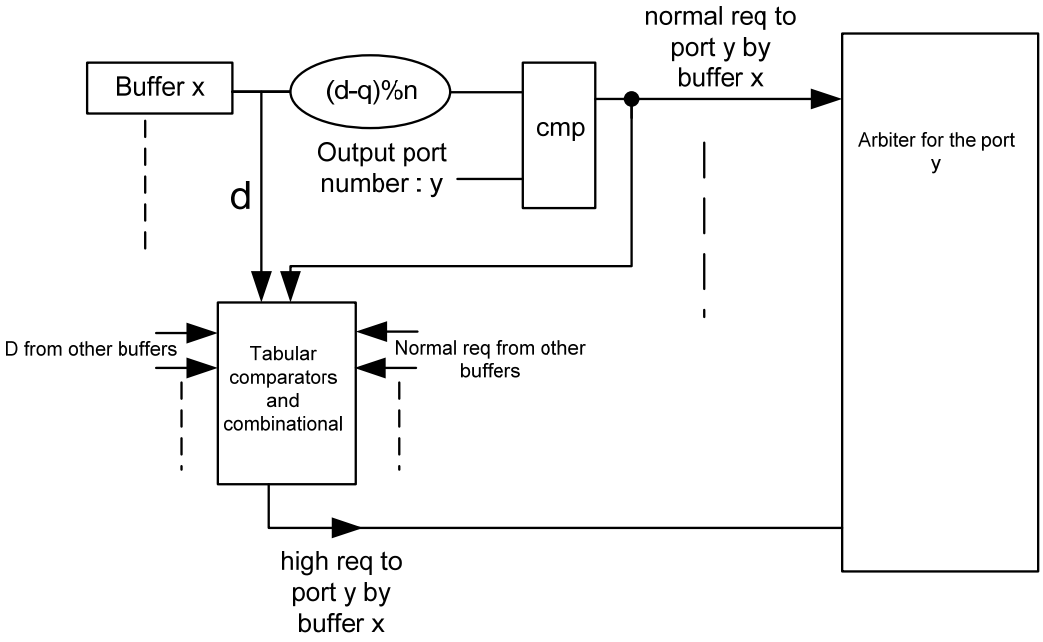


Figure 5-5 Two-Hit Filter Routing

III. Two-Hit Generation

Each two-hit generation subunit consists of four modules: an address unit, an update unit, a counter frame memory unit, and a bit vector memory unit. The update unit receives the arriving hits from routing FIFOs, reads the old hits on the corresponding diagonals from the counter frame memory, and generates the output that is written to the bit vector memory. Recall that the counter frame stores the coordinates of the most recent hit on a diagonal and has a length equal to the query length.

The memory units are mapped to FPGA's internal BRAMs. Each two-hit subunit stores one-fourth of the total counter frame and one-fourth of the total bit vector for a subject sequence. The address unit generates the read and writes addresses to the bit vector and counter frame memories.

- **Address Unit**

The bit vector and counter frame address from the input hit (d_x, q_y) are calculated as follows:

$$ADD = \frac{(d_x - q_y + q)}{4}$$

In which, q is the query length. Because the bit vector memory will be updated one clock cycle after the counter frame memory is read, a register is inserted between the ADD signal and the bit vector memory's write address.

- **Update Unit**

The update unit receives a new hit from the input FIFO and the most recent hit from the counter frame memory and generates the output bit for the corresponding alignment. The connections between the update units, address units, and the memories are depicted in the following figure.

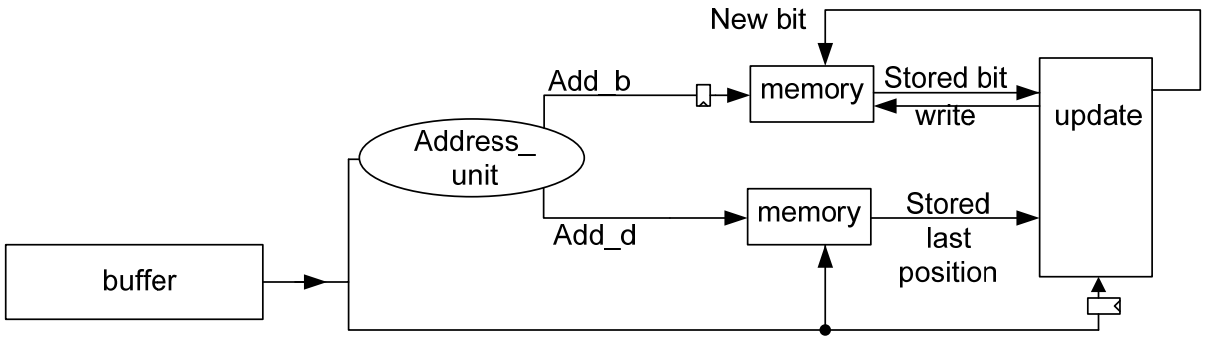


Figure 5-6 Two-Hit update Subunit

There are several issues that should be considered in the design of the update unit. In the following, we will describe them in a typical workflow of the update unit. The two-hit unit reads the new hit from the input FIFO and the old hit from the counter frame, respectively. Then, it checks if the hits belong to the same diagonal. If they are from different diagonals, that is, if $(d_{x0} - q_{y0})_{old} \neq (d_{x1} - q_{y1})_{new}$, then the new hit corresponds to a new alignment mapped to the same address in the counter frame. This indicates that a new alignment should be mapped to this counter. In this case, the counter is updated with the new hit and a 0 is written to the bit vector memory to start the processing of the hits in this alignment. If they do belong to the same diagonal, the two-hit conditions are checked. If the conditions are met, a 1 is written to the bit vector memory. Otherwise, the counter is updated, and no action will be taken in the bit vector memory.

Two observations are critical to the determining accuracy of the two-hit filter. First, the counter frame memory can have random matches generated from the previous sequences. These random matches can cause extra two-hits to be committed. In order to avoid these, a sequence ID is attached to the hit packet and is written to the frame counter. The content of the counter is only considered valid if the sequence ID of the old hit matches the sequence ID of the new hit. Otherwise, the counter is updated and a 0 is written to the bit vector memory. We noticed that a 4 bit sequence ID gives enough accuracy to this purpose. Second, the overlapping hits should be managed properly; otherwise, the number of the seeds that will be reported will be significantly more than what is required. Recall that, in order to generate a two-hit seed, the two matches under consideration should not overlap. We take the following approach: if the old and the new hit overlap, the counter is kept unchanged. In this case, we might miss a two-hit if there

is a third hit that doesn't pair with the old hit to generate a two-hit while still generating a two hit with the new overlapping hit. In order to avoid this, we extend the distance checking of the two-hit filter by 3. Our results show that the increase in the number of 1s in the bit vector due to this is negligible.

IV. Bit Vector Output Unit

The output unit provides the output interface of the two-hit units. The details of the functionality of the output unit differ from system to system. Currently, we have two architectures for this subunit. One architecture reads the contents of the four bit vector memories and outputs their contents consecutively as soon as the bits are committed to the BRAMs. The other architecture only provides a status bit indicating that the contents of its bit vector memory are valid until all of it is read out. Both of these architectures reset the bit vector memory while reading its contents. This way, we prepare the bit vector memories for the next subject sequence.

The output unit has three states: the starting, processing, and flush state. In the starting state, the subject index is less than query and subject length. Thus, the output unit doesn't commit anything. In the processing state, wherein the subject index is less than subject length but more than the query length, upon any increase in the subject index, the output unit can commit one bit. In the flush state, where the subject index is equal to the subject length, the output unit outputs the remaining bits. During this time, nothing is written to the BRAMs. The output unit uses counters to count the number of bits read out. For each subject sequence, the control characters inserted into database stream help us count the length of each subject sequence and calculate the required bits.

Note that an incorrect insertion or omission of a bit will change the entire bit vector, and, therefore, excessive care should be taken to avoid such glitches.

Compared with the seed generation module that is implemented in Mercury BLASTp, our two-hit filter has a significant advantage. Our two-hit filter does not use heuristics and, therefore, has exact agreement with the two-hit seeding algorithm used in NCBI BLAST.

5.4 EUA Filter

Recall that the key idea behind TreeBLAST is that an ungapped alignment can be performed with iterative merging using a tree structure that forms a two-dimensional systolic array. The database sequence is streamed across the leaves of the tree and one complete score sequence (the set character-character match scores for that alignment) is generated every cycle. The score sequences are processed by the tree, which is also pipelined. For each alignment, the score of the best local alignment emerges after a few cycle delays. The nodes of the tree consist of some basic comparison logic; the tree size is generally limited by the number of BRAMs on the FPGA and by the tree area for large queries. The structure can be modified in several ways to run more efficiently and to handle various cases.

Folding. To handle queries larger than can fit on a single chip, the tree is "folded". Rather than generating a scoring sequence every cycle, *i* cycles are required, where *i* is the number of folds. On each clock cycle, 1/i of the score sequence is generated. That is, the tree is used on multiple iterations to handle the sequence.

Replication. When queries are small enough to fit multiple trees on a chip, they are replicated to take advantage of available resources.

The idea behind EUA filter is to couple the database stream with a bit vector indicating which alignments can safely be ignored (as generated by the two-hit filter). For example, a 1 in the bit vector corresponds to a position where the alignment must be processed, and a 0 corresponds to a position where it can be skipped. We now look at the skipping mechanism.

5.4.1 Theoretical General Skipping

The idea behind general skipping is, on every cycle, to look ahead in the bit vector to find the next 1 (corresponding to the next alignment to be examined) and then slide the database the correct number of positions. Ideally, general skipping takes only the number of cycles equal to the number of ones in the bit vector. The additional hardware required, however, is complex. For a bit vector, the “look ahead” logic is similar to a leading one detector used; e.g. in a floating point adder. On each cycle, both the bit vector and the database stream must be able to , slide any number of positions up to the maximum number supported. This, in turn, requires that each register in the stream buffer have a multiplexor (MUX) that is large enough for every possible number of positions that could be skipped. It also requires complex routing logic. As a result, support for even a small range of choices makes the logic for general skipping more expensive than the original tree.

5.4.2 Skip-Fold Mechanism

Recall that an F-times folded tree is folded to 1/F its original size and requires only 1/F the logic of the original but requires F cycles per alignment rather than 1. The idea behind folded skipping is to process unfiltered alignments in F cycles (as before) and to process the others in only one cycle. The control for this scheme is thus extremely simple; i.e., there is no need for complex look-ahead or routing logic. Rather, if the bit-vector value of an alignment is a 0, the database stream simply needs to be shifted; if the value is a 1, then the filter will continue processing the alignment for a delay of another F − 1 cycles.

The hardware cost is a slight increase in control complexity; no other additional logic is needed. The performance benefit of folded skipping can be demonstrated as follows. Assume that the bit vector for a size N database has O 1s. Without skipping, an F-folded tree requires roughly F × N cycles to process the database. With skipping, the number of cycles is N + O × (F − 1). If F is 16 and N/O is 20, then the speedup is greater than 9×. This speedup occurs independently of the distribution of 1s in the bit vector. The question is: why bother folding at all? The answer is that folding gives a way to make the EUA structures (trees) substantially more compact than previously and, thus, allows them to be replicated. For example, a database of size N and a query of size M is handled by a single tree (with a single database stream). This takes N cycles. Now, replace the tree with F trees folded to 1/F their original size. This new structure now collectively support F database streams, each of which has a throughput that is a substantial fraction of the original. The limit on the number of trees is generally given by the query size (M), the number of folds (F), and the number and size of the block RAMs.

Efficient implementation of Skip-Fold idea depends on the implementation of the Folding mechanism. A naive approach to implement Folding is to store the results of the root node in a smaller tree as shown in Figure Figure 5-7-a.

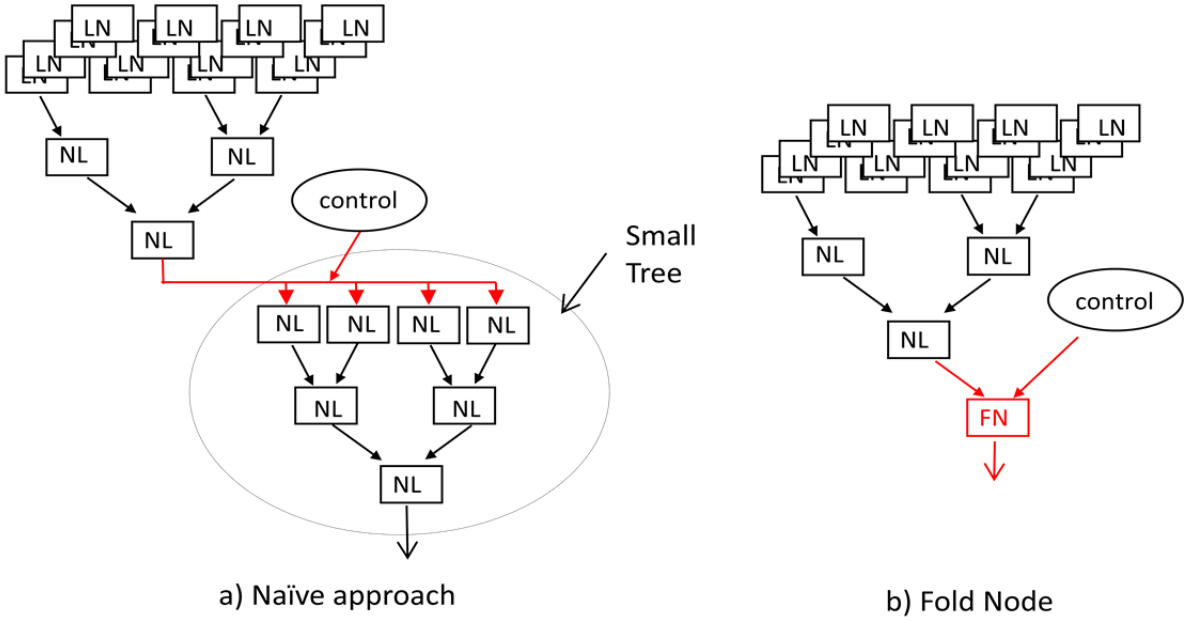


Figure 5-7 Implementation of a 4x Folded tree

The problem with the naive approach can be seen in the figure as well. The size of the small tree depends on the number of Folds and can become bigger than the original folded tree. In order to solve this problem and implement folding effectively, we have introduced another node type which we call the Fold node. Using the Fold node, the tree in Figure Figure 5-7-a is implemented as shown in Figure Figure 5-7-b. The fold node implements a sequential logic version of the NonLeaf node. In contrast with NL node, it only has one input set. It uses its internal running score set instead of the second input. At each clock cycle, it merges its input with its internal running score values(which are initialized to zero), and then updates the running scores. After F

clock cycles, it returns the running scores and resets them to zero. Using the Fold node, the implementation of the skip-fold mechanism and its control logic becomes extremely simple. If the input 2-Hit bit is zero, the state machine that controls the Fold node stays in initial state. If the 2-Hit bit is one, it takes additional F-1 clock cycles to return to initial state and return the result.

5.4.3 Seed Lookup Mechanism

The drawback of folded skipping is that, whereas 0s are processed F× as fast as 1s, they still take one cycle per character. Because the fraction of 0s (Z) is generally 98% to 99% of the stream, processing these null alignments still takes $\frac{Z}{Z+(1-Z)\times F}$ of the cycles, or 75% to 85% for almost all query sequences. The idea behind seed lookup is to limit the number of positions that can be skipped to a single number (S) (i.e., 16) that is determined experimentally. That is, the database stream skips either S positions or none. If there is a sequence of S or more 0s, then S skipping is used; otherwise, it is not. This scheme greatly simplifies the MUX logic. This idea alone will not be as beneficial as the skip-fold mechanism, but there is another idea that makes constant skipping extremely beneficial.

During the F clock cycles required by the skip-fold mechanism when EUA is working on an unfiltered alignment, a seed lookup module can stream in the database and the bit vector until it finds the next unfiltered diagonal. The seed lookup module finds the next unfiltered alignment by implementing a constant skip mechanism with S = 16. That is, during each clock cycle, it either skips one character or 16 consecutive characters until it finds the next unfiltered alignment. If the constant shift amount is set to 16, with a

typical F = 16, 16 × 16 = 256 filtered diagonals (0s) can be skipped. The performance gain is dramatic; only a small number of cycles are spent processing 0s, improving performance of this phase by more than 4×. Note that variable folded skipping addresses another significant issue with the EUA filter; i.e., the need to process artifactual null alignments that are inserted as padding during startup and teardown of each database sequence.

5.5 CAAD BLAST Architectures

We have implemented the CAAD BLAST filtering system in two different ways: multiphase and pipelined systems. In the multiphase system, each phase consists of one filter with the intermediate results stored in off-chip memory. We replicate each filter as much as possible. Operationally, we load the FPGA with a filter type, generate the filtering results using that filter and save the results in the external memory. Once done, we load the FPGA with the next filter. This way, we can replicate the cores maximally. Because the system consists of three filters, we have to reprogram the FPGA three times per run (four times if done in succession).

When originally conceived, we believed the multiphase algorithm to be preferable to the pipelined algorithm described next. The reason for the change is due primarily to two factors. First, configuration time has become less of a priority in recent FPGA designs than previously. Whereas a few years ago FPGA configuration took only about a tenth of a second, it now takes well over a second. Some of this time is due to the FPGAs themselves being larger, but more important is the commercial decision not to use board-level resources on the capability of fast configuration. The second reason is that

we originally overestimated the complexity of the pipelined implementation. While extremely challenging, it has proved plausible in the time budget of this dissertation.

Our goal now in presenting the multiphase design is to assess the area and performance of the system and lay out the foundation for the pipelined system. In the pipelined system, all three filters are chained together in a single architecture. Thus, there is no need for reprogramming. On the other hand, the granularity of the cores decreases, and, therefore, the replication factor will not be maximal. Also, delicate load balancing is required to make sure that no unit is overloaded. We will discuss both designs in the subsequent sections. As always, changing technology or commercial priorities may make one or the other method preferable in future FPGA generations.

5.6 Multiple Phase System on a Gidel Board

5.6.1 System Configuration and Operation

In the multiple phase system CAAD BLAST operation is as follows. For each filter, the FPGA is configured, the sequence is loaded and the filters are executed. In the first phase the two-hit filter generates a bit vector that is stored in the on-board memory. In the second phase the EUA filter reads the bit vector and the database (a second time) and returns a list of high scoring sequences. It saves their addresses to the onboard memory; we refer to this reduced database as DB'. In the third phase a Smith

Waterman filter is run on DB'. Processing continues with the Smith-Waterman filter until DB” is generated. In the final step, DB” (or DB’ for ungapped alignment) is formatted and executed with the original NCBI BLAST.

To accomplish this, two problems need to be solved. The first is to get agreement right. There are two parts: determining the internal thresholds that NCBI BLAST would use, especially cutoff, and correctly computing the E-values in the final report. The second and more serious problem is that we need to ensure that DB” both (i) contains all the sequences that NCBI BLAST would return, and (ii) is sufficiently reduced so that the overhead of formatting DB” does not overwhelm any potential performance gain. Figure 5-8 shows the global structure of CAAD BLAST.

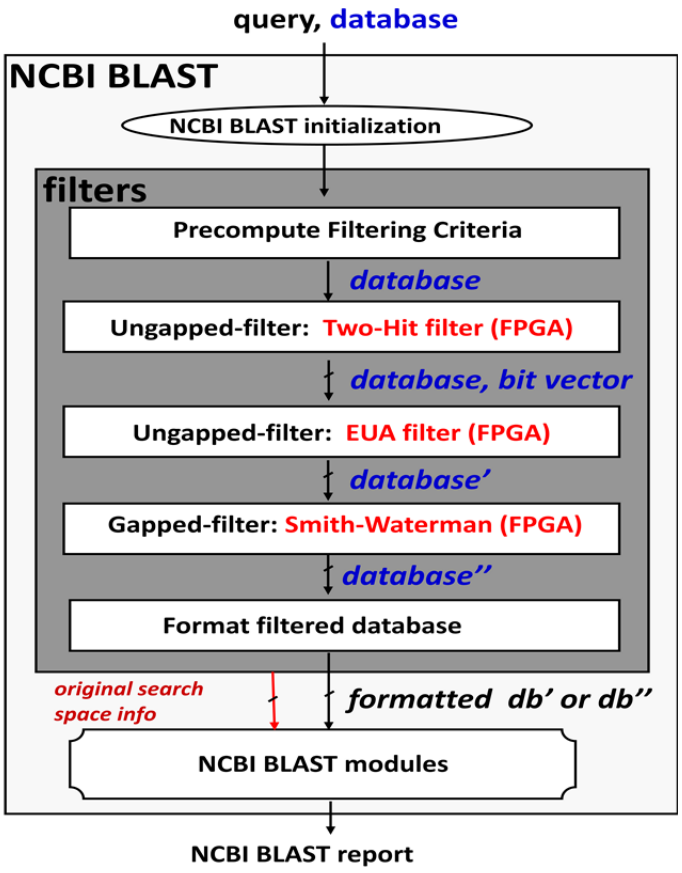


Figure 5-8 CAAD BLAST

In the precompute module, the host uses logic from the NCBI code to compute the various parameters needed to determine cutoffs and Evalues for both ungapped and gapped options. To ensure that the Evalues match those that would be computed by the original code, we also pass the original search space information. We have implemented all three filters on the reference system which contains a Gidel PROCe III FPGA board. The FPGA is an Altera Stratix-III 260E. At the time of implementation, this was a high-end device (using the 65nm process), but now is nearly three generations old. For on-board memory there is 4.5GB of DRAM partitioned into three banks of 2GB, 2GB, and 512MB, respectively. Each bank has a 64-bit interface and

can be accessed independently. One of the 2GB and the 512MB banks run at 333MHz; the other 2GB bank runs at 166MHz.

We have written a daemon code that keeps the database preloaded into staging memory. Unlike NCBI BLAST the database is unformatted. The user specifies the query and parameters using the NCBI BLAST interface.

While we currently assume a high-end FPGA, CAAD BLAST is easily decomposable and also runs well on low-end devices. Assuming sufficient memory bandwidth, the performance is roughly proportional to the number of BRAMs. The size of on-board memory should be sufficient to store the database.

5.6.2 Results

For the 2-hit filter, performance depends on the number of filters which, in turn, depends on the FPGA resources needed for each filter instance. The logic required is trivial, consisting of less than 1% of that available on the reference FPGA. The on-chip memory required, on the other hand, is the critical resource.

Table 5-2 Two-Hit Filter Statistics

Query Size	# of 2-Hit Filters	# of Hits per DB char	# of excess cycles per DB char
81	38	0.064	0.0002
217	35	0.205	0.0100
490	28	0.567	0.0524
838	25	0.891	0.2203
1204	21	1.244	0.3062
2205	14	2.570	0.8790

Table 5-2 Two-Hit Filter Statisticsshows the number of two hit filters that can be instantiated, using the design described on a high-end Stratix III FPGA, for a selection

of sequences from the NR database. The primary design decision therefore has to do with the structure of the position table, in particular the number of positions per entry in the primary table. For most query sizes (less than 1K) this number (3) falls out immediately from the convenience of packing that number of 10-bit addresses into a single 32-bit word. Also for small queries, having, say, 100 filters does little good: that is far more than the number of streams that can be supported by the memory interface in the reference design. For larger queries, there is the possibility of optimization by trading off table size for number of filters. That is, by having more entries in the primary table, some accesses to the secondary table can be avoided. But the larger table size allows fewer filters to be fit on the FPGA and so fewer database streams to be processed in parallel.

The right two columns in Table 5-2 give an indication of this trade off. The number of hits per database character (3-mer) is independent of the structure of the position table. For queries of size 1K, the expected number of hits per position is only slightly more than 1; having three positions per entry allows the primary table to account for most 3-mers. For the query of size 2205, however, the secondary table must be accessed frequently. The rightmost column illustrates this: it shows the number of excess cycles per database character; i.e., the number of extra cycles needed due to accessing the secondary table. For small queries, there are virtually no excess cycles, but for the 2205 query, nearly half the cycles are due to secondary table accesses.

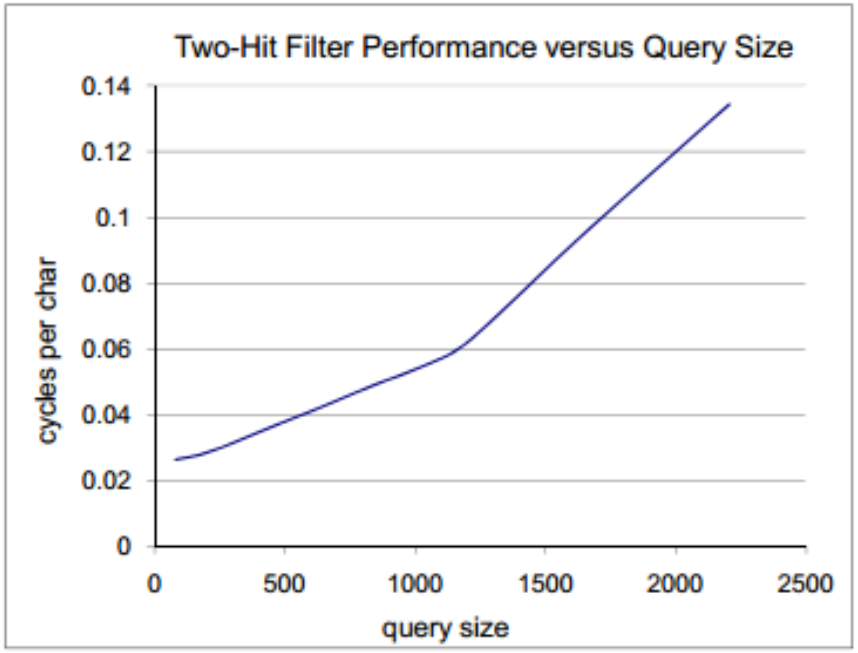


Figure 5-9 Graph shows performance as a function of query size for the Two-Hit filters in the reference design.

The performance of the 2-Hit filter phase depends substantially on the query size. There are two effects: the number of filters per chip and the amount of throttling that needs to be done because of references to the secondary table. Experimental results are shown in Figure 5-9 in terms of cycles per character as a function of query size. For typical protein sequences, size 100 to 500, the throughput is at least 25-30 characters per cycle.

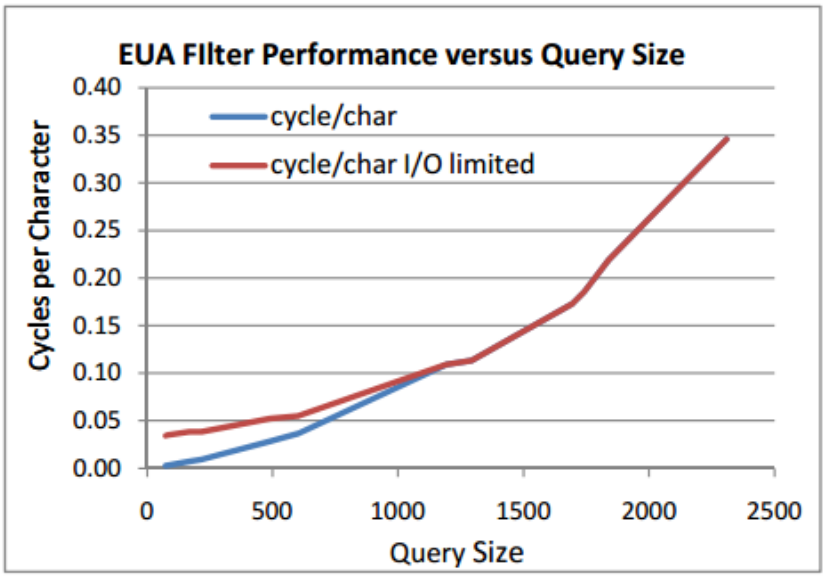


Figure 5-10 Performance as a function of query size for the implementation of the EUA filter in the reference design.

For the EUA filter phase, the limit on the number of trees (filters) is generally given by the query size M, the number of folds F, and the number and size of the BRAMs. For the reference design, the number of columns of the scoring matrix that can fit in an M9K BRAM is 32. Since BRAMs are dual ported, it is most efficient to use them to look up two characters at a time. This places a practical limit of 16 on F. Given the 912 BRAMs in the reference FPGA, the maximum number of EUA filters is 1824 × F/M, or 96 for M = 300. The graph in Figure 5-10 shows performance in cycles per character as a function of query size. The upper graph assumes that the computation is memory bandwidth limited, the lower does not. For the “limited” graph, the range is from 3 to 30 characters per cycle.

Table 5-3 Various results for CAAD BLASTP. Averages from running sequences of NR versus NR.

NCBI BLASTP	Gapped
exec time on lab PC Exec time on web server	46s 12-20s
CAAD BLASTP	
NR' (sch1) reduction from NR % of sequences remaining % of residues remaining NR" (S-W) reduction from NR % of sequences remaining % of residues remaining Format overhead NR" NCBI exec. overhead NR"	 3.24% 6.04% 0.054% 0.088% 0.53s 2.62s

Table 5-3 contains various results from the filter and reference runs. Our primary reference system is a 2008 64-bit 3GHz Xeon quad processor (Harpertown X5412) with 8GB of memory. We have used NCBI BLAST 2.2.20 (legacy) for reference and for the base code of CAAD BLAST. We have implemented all three filters on the reference system which contains a Gidel PROCe III FPGA board. We compiled each with standard optimization settings and run with default settings. For additional reference we use the web server at NCBI.

We now discuss some of these results. We note that they are averages; there is variation as expected from sequences of widely varying sizes. A database sequence is retained if it contains at least one HSP that scores above the cutoff. The NR database is reduced by a factor of 17. For gapped processing with Smith-Waterman, NR is reduced by a factor of 1136 and generally only a few thousand sequences remain. The formatting overhead includes host processing for the filters.

Table 5-4 contains performance results of the reference design with respect to the NR protein database as of 2009. Also shown are results for the unaccelerated host PC and the NCBI Server.

Table 5-4 Performance of the reference design with respect to the 3.53G residue NR database.

Query size percentile	2-hit chars/cycle Time	EUA chars per cycle time	S-W and overhead	Total time accelerated	Total time CPU only	Total time NCBI Server
Up to 500 78th	25/Cycle 1.3s	20/Cycle 1.3s	3.8s	6.4s	26s	14s
Up to 1000 97th	18/Cycle 1.9s	11/Cycle 2.4s	5.3s	9.4s	46s	20s
Up to 2000 99.5th	7/Cycle 4.8s	4/Cycle 6.6s	7.8s	19.2s	95s	40s

For CAAD BLAST the S-W time is less than the time for the other filters. Most of the time is in executing the final run of NCBI BLASTP. By percentile we indicate the rough proportion of queries that are smaller than the size shown [Cou05]. Speed-ups over the unaccelerated PC range from 4× to 5×. The NCBI Server is a large cluster that processes queries in parallel according to load.

5.6.3 Scalability Analysis

One issue with FPGA-based systems is that no standards have been adopted as to the proper configuration of a system for high-performance computing (HPC) with FPGA coprocessors. This is in stark contrast to GPUs, where application developers and HPC system builders have a very good idea of what to expect. For FPGAs, on the other hand, there are wide ranges of both quantity and types of FPGA resources (i.e., on the FPGA chip). Even more critically, the supporting infrastructure is completely vendor-dependent and also varies widely. Exacerbating the problem, there are many such

vendors, and no one vendor has a dominant position in the market or is supported by either of the major FPGA producers. These board-level parameters include: the amount of onboard memory, FPGA-memory interface (total bandwidth, latency, number of streams, and flexibility of the streams), I/O bandwidth, and configuration time. These differences make it extremely difficult to predict performance for a given application, even starting with the best reports in literature. The purpose of this section is to allow users of high-performance, FPGA-based systems and potential developers of such systems to predict the performance of CAAD BLAST on those systems. Our goal in this section is to investigate the scalability of CAAD BLAST when mapped to different FPGAs. This is done in two dimensions: timing and area. With regard to both area and performance, we have calculated theoretical models that show how the system scales on different systems. We have based our study on Altera FPGAs and our multiphase system.

CAAD BLAST consists of four phases:

T_2h = Time to run Two-hit Filter

T_tb = Time to run Exhaustive Ungapped Alignment Filter

T_sw = Time to run Exhaustive Gapped Alignment Filter

T_ncbi = Time to run NCBI BLAST

There is overhead between the phases. This primarily affects the first three (the FPGA) phases and consists of the time for FPGA reconfiguration before the start of the phase T_config. There is also some miscellaneous overhead (T_misc), including the time required to compute the contents of the query-specific data structures, load the FPGAs with data, and format the database for the final NCBI BLAST pass.

T_total = T_2h + T_tb + T_sw + 3*T_config + T_ncbi + T_misc

FPGA Phases in General

Each FPGA phase consists of streaming the database through the FPGA in some number of streams and outputting some amount of data.

In each term, T_2h, T_tb, and T_sw depend on three things:

1. The database size (Size_db)

2. Streaming bandwidth

3. Various additional factors related to the efficiency with which the bandwidth can be used

The streaming bandwidth can be limited either by the external bandwidth of the memory interface (BW_mi) or by the internal bandwidth of the processing configurations (BW_2h, BW_tb, and BW_sw, respectively, for each phase). The bandwidth of the memory interface BW_mi can be limited by the FPGA's I/O BW capacity, but, generally, FPGA-based systems are constructed to not use more than a fraction of that capacity. The FPGA I/O BW, not including the high-speed serial interfaces, is at least 20 GB/s for any high-end chip produced in the last 5 years, most being much higher. Our Gidel board, however, has a memory bandwidth of 333 Mhz * 16 B + 167 Mhz * 8 B = 6.7 GB/s. In this application, 333 MHz * 1 2B = 4 GB/s is usable to stream the database. The internal bandwidth for each configuration is related to the number of parallel filter units and the operating frequency of those units.

For the various phases:

• P_2h, P_tb, P_sw = various numbers of processors/streams

• F_2h, F_tb, F_sw = various operating frequencies

Both of these terms, the number of processors, and the operating frequency, depend on the FPGA resources and the process generation. For each phase, the number of processors also depends on the query size (S_query). The resource requirements are in three categories: BRAMs, logic, and I/O bandwidth. The different phases use these in different proportions, so the limiting factor varies from phase to phase. With respect to the I/O interface (off-chip memory), substantial overhead logic is required to interface to the large number of streams possible, especially for the two-hit filter. For example, in the Gidel system, the database can be partitioned across two memory banks, each of which has a 64 bit interface. This physical bandwidth can be translated into a number of virtual streams by the interface logic (16 for the Stratix-III). These virtual streams are constructed automatically using vendor tools and can take up to 20% of the FPGA's logic and also a number of BRAMs. Various additional factors either speed up or slow down the processing, such as:

- The amount of data that must be output. One could imagine this cutting into the external bandwidth capacity. However, with the most recent implementations, the output stream is small for all phases.
- Speedup and slowdown factors. These are mostly algorithmic, complex, and phase-specific. These can be substantial and are described in detail below. Because the replications are totally independent, there is no problem in routing and mapping.

Scaling to future-generation FPGAs, various FPGA families, and FPGAs of various vendors. We give the performance numbers as functions of various resource

capabilities. For logic and components (BRAMs), this is straightforward. For chip I/O bandwidth (BW_mi), we make the following observations:

- The glue logic for the Gidel system described takes up to 20% of a Stratix III, 10% of a Stratix IV, and a smaller percentage of a Stratix V.
- An alternative way of looking at this is to keep the fraction of logic fixed at 20%. In that case, the bandwidth supported doubles in each generation.

Phase 1: Two-hit filter -- T_2h

The number P_2h of units depends on the resources available on the device and those required for the computation. Because the logic requirement is trivial, the BRAMs or BW_mi are the limiting factor. The number of BRAMs depends on the query size.

We use the following notation to parameterize the BRAM resources available and required:

T_144 : Total number of M144k BRAMs available

T_9 : Total number of M9k BRAMs available

T_20 : Total number of M20k BRAMs available

RQ_144: required number of M144ks per query neighborhood for a given query

RQ_9 : required number of M9ks per query neighborhood for a given query

RQ_20 : required number of M20ks per query neighborhood for a given query

RA_9 : additional required number of M9ks per stream for internal calculation

RA_20 : additional required number of M20ks per stream for internal calculation

The following tables give the area usage of the two-hit filter and the corresponding replication number. The Stratix III and Stratix IV each have some mix of 9 K and 144 K BRAMs. The Stratix V has all 2 0K BRAMs.

Table 5-5 two hit filter area

Query size	RP_9	RP_144	RA_9	RP_20	RA_20	ALUT	ALM	Registers
256	50	3	12	25	9	1800	1425	799
512	84	6	12	42	9	1800	1425	799
1024	93	6	16	47	9	1800	1425	799
2048	136	8	24	68	13	1800	1425	799

The number of two-hit filters is also affected by the interface overhead. From our experience with the Gidel interface and the Stratix III, we estimate the following overhead for the Stratix family:

Stratix III: For 32 read and write ports, we need 270 M9ks

Stratix IV: For 64 read and write ports, we need 540 M9ks

Stratix V: For 128 read and write ports, we need 540 M20Ks

Including this overhead gives us the following replication sizes:

Table 5-6 two hit filter replication

Query Size	P_2h for Stratix III	P_2h for stratix IV	P_2h for stratix V
256	32	48	96
512	24	32	70
1024	20	32	64
2048	16	22	50

The maximum number of filters on the Stratix III and Stratix IV are:

$$P\_2h\_unlimited = 2 \times \frac{T\_144}{RQ\_144} + 2 \times \frac{T\_9 - 2 \times \frac{T\_144}{RQ\_144} \times RC\_9}{RQ\_9 + RC\_9}$$

On the Stratix V, the maximum number of filter is:

$$P\_2h\_unlimited = 2 \times \frac{T\_20}{RQ\_20 + 2 \times RC\_20}$$

The derivation is as follows. In general, this is the total amount of the resource divided by the amount needed per unit. For the Stratix V, the corrections are due to the fact that BRAMs are dual-ported and can be used for two streams (RQ_20) or not (RA_20). For the Stratix III and Stratix IV, the additional complexity is due to there being two ways to construct units: (i) out of M144Ks for RQ and M9Ks for RA or (ii) out of M9Ks for both RQ and RA. The left term has the RQ part of (i) while the right term has the RA parts of both methods and the RQ part of (ii).

At this point, we could naively compute the time as T_2h = S_db / P_2h*F_2h, but there other limiting factors, such as:

1. The bandwidth of the memory interface BW_mi might be less than P_2h*F_2h.
2. In our current multiphase implementation, we need to save a bit vector in off-chip memory. This also consumes BW_mi. However, in our current scheme, this output bit vector is heavily compressed, so this effect is negligible.
3. Only one DB sequence is allowed to be evaluated at a time by a single two-hit filter. That is, there can be no overlap among DB sequences. Thus, it takes roughly 3 x S_q cycles to process a DB sequence because there need to be three roughly equal-sized phases; i.e., startup, steady-state, and teardown.
4. Less than one character per cycle can be looked up because of the need to go to the secondary table. On the basis of our experiments, the weighted average of

the number of clock cycles spent on the secondary table is less than 10% of the total time calculated above. However, this number, depends on query data and query size; it has both high variance and increases drastically with large queries. Sample results from 30 queries are as follows:

Table 5-7 average of the number of clock cycles spent on the secondary table

Query Length	Average	Max	Min
1-256	1.6%	8%	0%
256-512	7%	16%	1%
512-1024	32%	56%	14%
1024-2048	128%	160%	81%

Thus, an estimate of the two-hit filter running time, assuming that two-hit filter throughput is limited by the internal bandwidth, is:

$$T\_2h = \frac{2 \times S\_db + S\_q \times S\_db\_\#\_Seq + S\_extra}{P\_2h \times F\_2h}$$

*Where d_extra=table_value * S_db*

Fixing the extra cycles at a conservative 1/3, the estimate becomes:

$$T\_2h = \frac{4}{3} \times \frac{2 \times S\_db + S\_db\_\#\_seq \times S\_q}{P\_2h \times F\_2h}$$

We now need to go back and see where we will be limited by BW_mi.

BW_2h = S_db/T_2h

For the Stratix III, Stratix IV, and Stratix V, the average numbers of P_2H are 24, 32, and 70, respectively. Then, the internal bandwidths are 1.3 GB/s, 1.8 GB/s, and 3.9 GB/s, respectively. All of these are less than the raw available bandwidth of our Gidel board and a small fraction of the of the FPGA's capacity:

$$T\_2h = \frac{S\_db}{P\_2h \times 56Mhz}$$

Phase 2: Exhaustive Ungapped Alignment filter (T_tb)

For the EUA filter, the replication of filter units is logic limited. As a result, the total number of ALMs on the FPGA is divided by the total required ALMs per stream in order to generate the replication size estimate.

$$P_{tb} = \frac{\text{total ALMs on FPGA} - \text{ALMs used for memory interface}}{\text{ALMs required per stream}}$$

The following tables give the resource usage of the EUA filter and the corresponding replication number. This includes reserving 20% of the logic for the memory interface.

Table 5-8 TreeBLAST Area and Replication Number for Stratix III and Stratix IV

Query Size	ALUT	ALM	Registers	M9k	P_tb for Stratix III	P_tb for Stratix IV
1-255	5817	4670	4938	8	16	32
256-511	9574	7815	9113	16	11	21
512-1023	17065	14287	17474	32	6	14
1024-2047	32179	27365	34203	64	3	6

Table 5-9TreeBLAST Area and Replication Number for Stratix V

Query Size	ALUT	ALM	Registers	M20K	P_tb for Stratix V
1-255	5358	5781	4665	8	49
256-511	8714	9848	8650	16	32
512-1023	15502	18214	16627	32	17
1024-2047	29059	34858	32588	64	11

For EUA, there are two additional factors that affect stream throughput:

1. Stream padding. Database sequences should not overlap. Therefore, S_q null characters ($) are inserted between database sequences. Consequently, the number of characters that must be streamed is the size of the database plus the number of sequences in the database times the query size: S_db + S_q *S_db_#_seq.
2. Fraction of alignments that pass the two-hit filter of phase 1. With the default folding factor of 16, each "passed alignment" takes 16 cycles. Almost all of the remaining latency is hidden; i.e., it skips the "nonpassed alignments." By convention, we use 1s to signify passes and E_1 to express the ratio of all set bit in the bit vector to total bit vector size. E_1 varies from 2% to 5% depending on query size and query composition. The weighted average is close to 2% (see Table 9 below). Note that these two factors are correlated. Although it is annoying to need to pad the TreeBLAST filters with null characters, most of the latency is hidden with the skip mechanism.

Table 5-10 E_1 versus Query Size

S_q	average	max
256	0.008	0.009
512	0.016	0.028
1024	0.02	0.025
2048	0.027	0.043

In general, the upper bound on P_tb can be calculated as follows:

$$P\_tb = \min(\text{P\_tb\_unlimited}, \frac{\text{avg\_seq} + q\_size}{avg\_seq} \times \frac{off\_band \times E\_1 \times 16}{f\_tbf})$$

If the throughput of the TreeBLAST filter is limited by the internal bandwidth, then:

$$T\_tb = \frac{(S\_db + S\_q \times S\_db\_\#\_Seq) \times E\_1 \times 16}{P\_tb \times F\_tb}$$

Again, we now need to go back and see where we will be limited by BW_mi:

$$BW\_tb = S\_db/T\_tb$$

For the Stratix III, Stratix IV, and Stratix V, the average numbers of P_tb are about 10, 20, and 30, respectively. Then the internal bandwidths BW_tb are 6.0 GB/s, 12.0 GB/s, and 18.0 GB/s, respectively. All of these are greater than the usable BW_mi for that generation, which is 4 GB/s for the Stratix III and estimated to be 8 GB/s and 16 GB/s for the Stratix IV and Stratix V, respectively. Therefore, for current FPGA coprocessor designs, the true T_tb is likely to be:

$$\text{T\_tb} = \frac{S\_db}{BW\_mi}$$

Phase 3: Exhaustive Gapped Filter with Smith-Waterman (T_sw)

For the exhaustive gapped filter pass, we use Smith-Waterman. Because the database has been heavily reduced by the previous phases, little effort has been made so far to parallelize or otherwise optimize here. Therefore, we assume a single filter which can be folded as needed for large sequences. The following table shows various statistics. The number of folds required is ceil(S_q/MaxQuerySize).

Table 5-11 Smith-Waterman

ALUTs per PE	223
ALUTs per PE w/Folds	227
ALMs per PE	140
ALMs per PEw/ Folds	148
Registers per PE	63
M9Ks per PE	1
M144Ks per array w/Folds	16
Stratix III-- Max query size w/o folding	650
Stratix IV- Max query size w/o folding	1450
Stratix V- Max query size w/o folding	2500

The following table shows the average ratios derived from experiment:

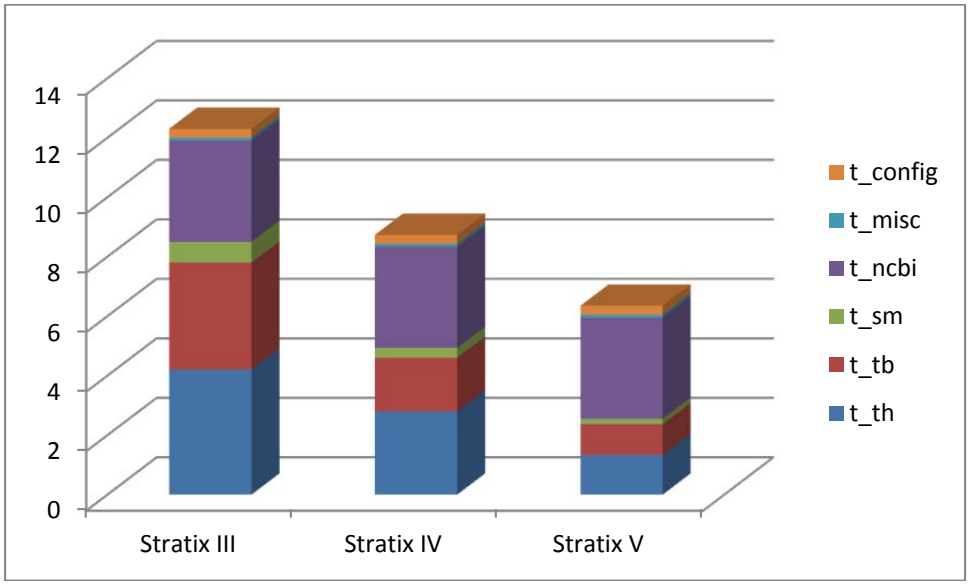
Table 5-12 Fraction database remaining after TreeBLAST and before Smith-Waterman phase

S_q	S_db' in number of chars	S_db' in number of sequences
256	0.01	0.02
512	0.03	0.04
1024	0.05	0.09

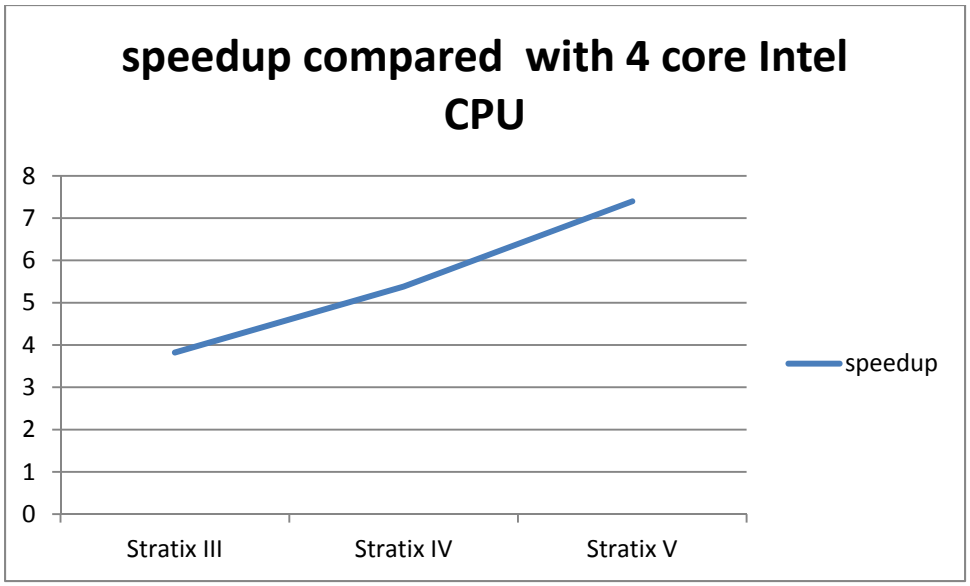
Assuming that we are limited by the internal bandwidth BW_sw, the time per query is:

$$T\_sm = \frac{S\_db \times rd1\_char + S\_q \times S\_db\_\#\_Seq \times rd1\_seq}{F\_sw}$$

where rd1_char is the fraction of the database remaining in characters, and rd1_seq is the fraction of the database remaining in number of sequences.



5-11 Expected Timing results based on timing model of CAAD BLAST on different FPGAs



5-12 Expected speedup results based on timing model of CAAD BLAST on different FPGAs

Figures 5-11 and 5-12 show the expected timing and speedup on three generations of FPGAs. The calculations are based on the timing model described before and the actual runs on the Gidel board. The timings are based on a version of the NR database with 5.5G characters and 15.6M sequences. As can be seen the FPGA time is halved from one generation to the next. This is obviously expected since the FPGA resources double over time. On the other hand the remaining timings do not change. This makes the final NCBI BLAST run time a bottleneck in the latest generations. As expected the speedup linearly increase from one FPGA to the next generation FPGA.

5.7 The Pipelined System on a Convey Machine

Theoretically, from a parallelization point of view, the multiphase system is an ideal solution, the reason being the fully parallel nature of the filters. For all three filters, the workload can be distributed between replicated modules with the only overhead coming

from the distribution and support of the multiple streams. The modules themselves can be replicated as many times as possible, taking advantage of all the available resources on the chip. The major problem with this approach is the time needed for reprogramming. Although FPGAs generally support programmability in milliseconds, most commercial boards need at least a full second to program the FPGA. This extra overhead slows down the original application. Thus, we decided to chain all filters together. In this Section, we introduce the pipelined system. The pipelined system is implemented on the Convey machine and its Xilinx FPGA.

5.7.1 System Configuration and Operation

A database server reads the database from a disk in Fasta format and creates the database data structures that are required by the hardware. The server then shares this data structure through a shared memory interprocessor communication mechanism with the client BLAST applications. Each protein residue is encoded as a binary value between 0 and 25. In order to indicate the end of a sequence, each database sequence is appended with a special control character. The control character is used by the hardware to separate the processing of subject sequences. Its binary encoded value is 26. The subject sequences are extended with dummy letters such that they are all multiples of 16 characters in length. This is required to simplify the operation of the hardware, particularly the 16x mechanism described earlier.

The database is organized in two main parts: sequence array and offset array. All of the subject sequences are concatenated together and stored in the sequence array. The sequences are separated with the special control character described above. The starting locations of the subject sequences are stored sequentially in the offset array.

For each subject sequence, the starting character's location, relative to the first character in the sequence array, is stored in the offset array.

Each EUA unit is responsible for processing a portion of the database. The partitioning of the workload between multiple FPGAs and multiple EUA units is performed with a data structure that we call the context array. Each FPGA has its own context. A context contains the following information: a pointer to the subject array, a pointer to the offset value that corresponds to the first sequence that should be processed by the unit, the number of sequences that should be processed by the unit, and a pointer to a memory location to store the generated results.

In order to retrieve the first sequence, the EUA unit adds the offset value of the first sequence to the subject array's address. Subsequently the EUA unit will read the subject sequences from the subject array until the required number of sequences is processed. The offsets are used to interleave and distribute the subject sequences among two-hit units.

5.7.2 Pipelined Filters

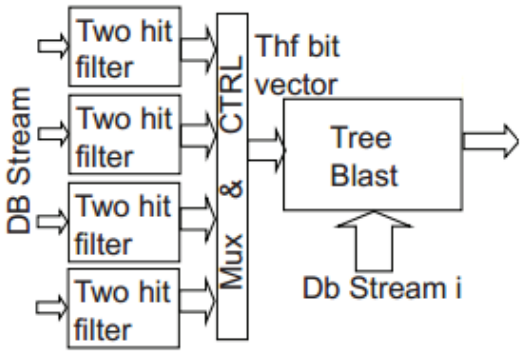


Figure 5-13 pipelined filtering unit

Figure 5-13 shows the overall scheme of a single-filter bank. Parallel database streams feed the two-hit filters, which, in turn, send the 0/1 stream to an EUA filter. A copy of the database is streamed to the EUA filter, where it is coupled with the 0/1 stream.

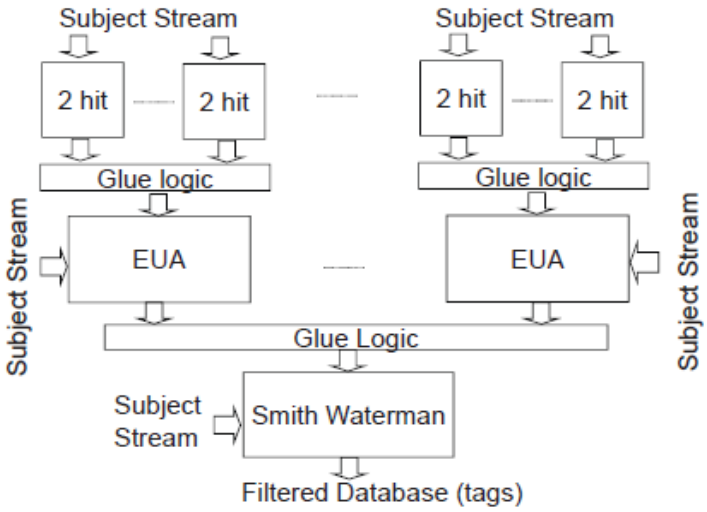


Figure 5-14 Block diagram of Accelerated BLAST

This structure is replicated a number of times depending on the size of the query and the FPGA. In the final stage, the highest scoring database sequences from all of the banks are processed with a single S-W module. Speed matching between the two-hit and EUA stages is accomplished as follows. The EUA filter processes data (a single sequence) from a single two-hit filter at a time. Processed sequences from the other two-hit filters in the bank are buffered. Through the mechanism described in the previous subsection, the EUA filter is capable of consuming three to five characters per cycle; that is, data from buffered filtered sequences are transferred to the EUA filter F characters at a time (in this study, F = 16). After processing the data of one two-hit filter, the EUA filter starts working on the next sequence from the next two-hit filter. In order to load balance, the database sequences are sorted by length and multiplexed

among multiple two-hit filters. As a result, the time required to process successive sequences is nearly equal.

Coupling with the S-W filter is accomplished as follows. For each database sequence, the EUA filter compares the maximum score generated with a constant threshold. If this score is larger than the threshold, the EUA filter writes the address of the sequence to a FIFO. The S-W unit reads these addresses, streams the subject sequences, and calculates the maximum scores.

5.7.3 Accessing On Board Memory: The Jump FIFO Interface

Throughout the multiple designs, FPGAs, and FPGA platforms, a uniform interface is used to access the external memory and retrieve the required data. Following the terminology of the Gidel IP library, we call module the jump FIFO interface. Its interface consists of the following five signals.

- Jump: requests a "jump" to a specific address in the memory,
- Address: the address of the memory to jump to, when the jump signal is set,
- Data: the data that is being transferred through the port,
- Read/Write: sequential read/write requests,
- Ready: Interface ready for the next transaction. This can be interpreted as data valid in case of reading, and output port ready to receive another data in case of write.

The jump FIFO interface is basically a FIFO interface, except that it has an embedded jump functionality. In case of reading for example, when jump is set, the FIFO should

load the data from the external memory from the address specified in the address port as soon as possible, and it should disable the ready signal until the new data arrives. There are several instances of this interface in the system. The jump interface was originally used and developed by the GIDEL company as part of their board's multiport memory controller IP core, and it proved to be a simple and efficient interface once we ported the design to the CONVEY computer. As shown in Figure 5-14, each EUA unit is connected to multiple two-hit units. For each EUA module, there is an address module that is responsible for interleaving the offsets among its two-hit units. As an example, consider a case when there are three two-hit units per EUA. In this case, while the EUA unit processes subject sequences sequentially, the two-hit filter *i* reads and processes sequences as $3k + i$ where k is an integer.

5.7.4 Glue Logic Modules

In order to simplify the design and streamline its reusability, we have implemented a module called stream_maker that, given a sequence of offsets, accesses the external memory through a jump FIFO interface and generates the character stream as if the offsets were not originally interleaved. stream_maker has a FIFO that is written by the address units and contains the offsets of the sequences that should be fetched. stream_maker generates the character stream which is fed to the two-hit filter. Similarly, once a sequence passes the EUA filter, its offsets are written to a stream_maker FIFO. The stream_maker generates the character stream for the S-W module, which, in turn, performs another level of filtering.

5.7.5 RTL Optimizations

For queries of up to 256 characters, accelerated BLAST (on each FPGA) consists of five clusters of two-hit/EUA filters, each with five two-hit filters feeding a single 16x folded EUA filter. In turn, these five clusters feed a single S-W filter (2x folding). Larger queries have analogous implementations.

The initial synthesis returned an unacceptably poor operating frequency. We tried reducing the design size, but the problem was not ameliorated until only an unacceptably small fraction of the potential chip capacity was in use. Instead, we solved this problem through two RTL mechanisms: floor planning modules with respect to BRAMs and redesigning the logic to reduce fan-outs and the lengths of the communication channels.

There are two problems that need to be dealt with through RTL-level logic: mapping function I/O to physical I/O and reducing path delay. These are both handled primarily through the creation of three modular communication interfaces: simple FIFO, jump FIFO, and a direct register-based interface. These interfaces are described further below. Using these interfaces, we can place each core anywhere on the FPGA and keep its communication off of the critical path by simply specifying an appropriate number of pipeline stages. Other optimizations include replicating registers to reduce fan-out and eliminating the reset circuit as much as possible. The simple FIFO interface serves as our flexible general purpose intermodule communication mechanism and is used especially to foster module independence and avoid the creation of long paths.

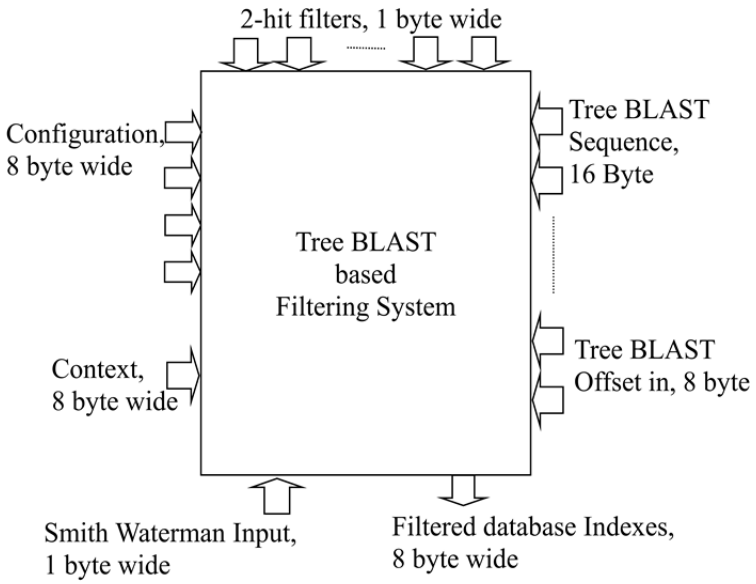


Figure 5-15 Accelerated BLAST external interfaces for

Figure 5-15 shows the external I/O interfaces for the accelerated BLAST configuration that supports sequences of length up to 256 characters. Note that there are 26 x 1B streams and 5 x 16B streams operating continuously and that there are a number of others that are used for initialization, data offload, and synchronization. These must be mapped to the physical I/O provided by the Convey HC-1ex: that is, the 16 x 4B memory channels that can operate independently at over 300 MHz. The mechanism we use is the jump FIFO interface described above.

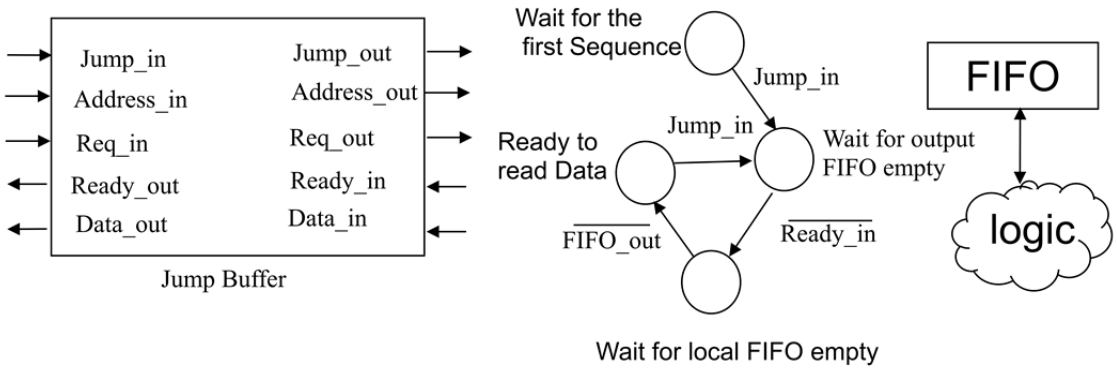


Figure 5-16 Jump IO Interface and Signals

A block diagram and a signals interface for the jump FIFO are given in Figure 5-16. It is a generalization of the simple FIFO in that it communicates with external memory at a specified address. The jump FIFOs are mapped to the Convey physical memory interface through the Convey memory crossbar module which routes memory transactions to the correct memory interface.

5.7.6 Replicating and Balancing the Components

We find the optimal number two-hit filters per EUA filter by measuring the fraction of idle cycles in the EUA filter as a function of the number of two-hit filters and the query size. The results are shown in Table 5-13 Balance between two-hit and EUA filters.and indicate that three to five two-hit filters per EUA filter is optimal.

Table 5-13 Balance between two-hit and EUA filters.

Query Size Fraction 1s	Ratio of 2-Hit to EUA Filter	EUA Filter Percent Idle Cycles
256 0.008	4	13
	5	**1.5**
	6	0
512 0.016	2	14
	3	**0.14**
1024 0.020	2	13
	3	**0.06**
2048 0.027	2	12
	3	**0.03**

Overall, the EUA filter enables the database to be reduced by at least 97% for most query sequences. Therefore, the S-W filter can be compacted substantially through folding and still obtain adequate performance. The optimally folded S-W filter consumes characters of the reduced database DB at the same rate that characters of the original database DB are consumed by the two-hit filters. The raw results are shown in Table 5-14.

Table 5-14 optimal number of folds in the SW filter

Query Size	Reduction db to db'	Number of Folds for SW Filter (Virtex6)	Number of Folds for SW Filter (Stratix V)
256	0.01	7	4
512	0.03	4	2
1024	0.05	4	2
2048	0.07	5	2

When integrated into the overall system, the number of folds is either two, four, or eight. From the preceding discussion, we see that a speed-matched bank of filters contains from three to five two-hit filters and one EUA filter folded to affect 16× replication. A single S-W filter is shared by all of the filter banks and folded to affect 2× to 8×

replication. The number of filter banks themselves that can fit on an FPGA is a function of query size and FPGA resources.

5.7.7 Floor Planning

We apply floor planning in two layers. The first layer is applied internally to the two-hit filters and the second layer is applied for the higher-level modules consisting of the two-hit filters that feed individual EUA filters. We found it sufficient to map BRAMs to particular modules and let the synthesis tools continue handling the logic placement. Although the two-hit filters each require only a small amount of area, their logic is complex and, more significantly, does not lend itself to pipelining. That is, pipeline stages would increase the time required to process each character, violating our most basic design constraint; i.e., flowing the database through the FPGA at a streaming rate of one character per cycle.

The critical path is the lookup of database w-mers in the query (see Figure 5-4). In the "fast" case, there are three or fewer matches in the query. In the "slow" case, there are more and a secondary table must be accessed. The complete two-hit filter is shown in Figure 5-3. Each fetched entry must be processed in one clock cycle, meaning that a newly computed address needs to be issued to the position list. As a result, the addressing circuit contains a combinational path that starts with the output of the position list and continues to the address input of the same position list.

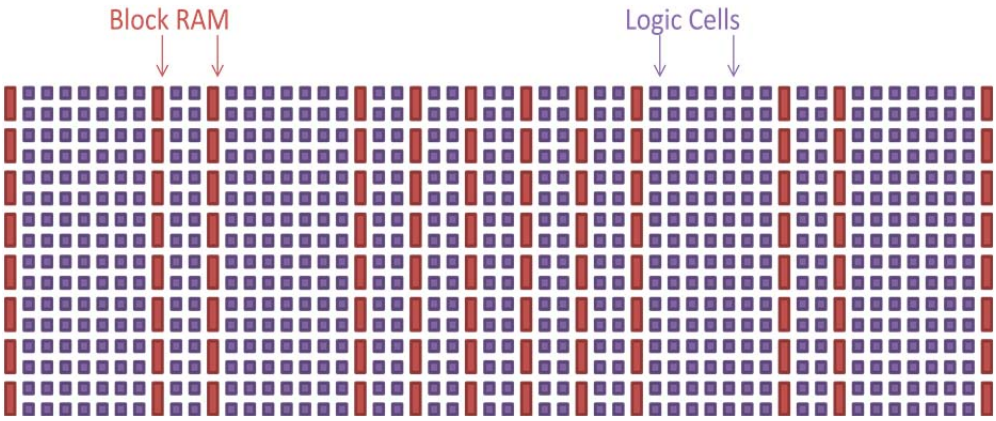


Figure 5-17 Cells in an FPGA

The FPGA consists of a pool of CLBs and BRAMS as shown in Figure 5-17. We number the BRAM columns from the left from 0 to 11. Of these, 4 to 7 are used by the interface logic and the API; this leaves 0 to 3 and 8 to 11 for user logic. To floorplan the two-hit filters, we place the BRAMs for the position lists in a square, minimizing the path length, as shown in Figure 5-18.

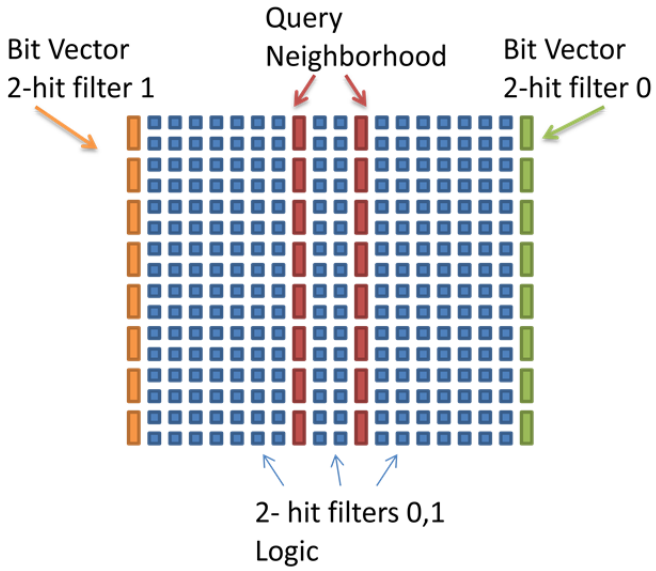


Figure 5-18 Two-hit filter after floor planning

At the next level, the EUA filter BRAMs are placed as close as possible to those of the two-hit filter (see Figure 5-19).

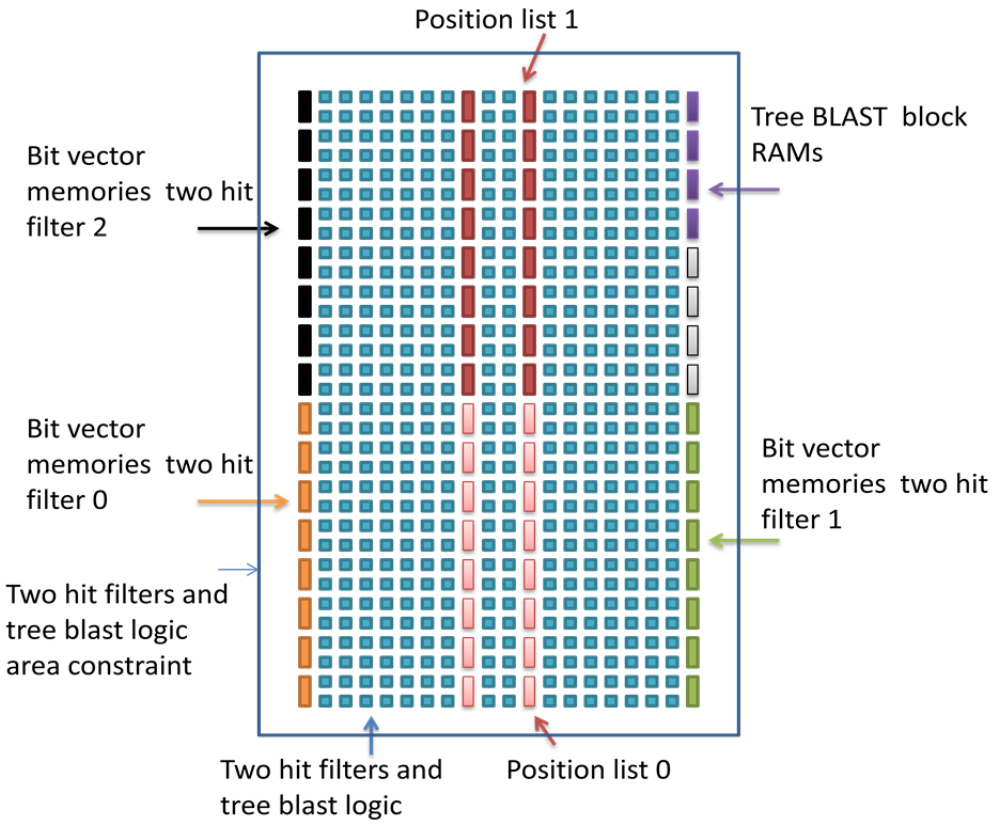


Figure 5-19 Module after floor planning

5.7.8 Integration and Results

Table 5-15 Per component resource utilization for the Alterashows the results for the Altera Stratix IV EP4SE820H40I3, and Table 5-16 shows the results for the Xilinx Virtex-6 XC6VLX7601. We find that the Stratix IV, depending on query size, can fit 5, 5, 4, or 3 filter banks for a total of 25, 15, 12, or 9 input streams. The Virtex-6 can fit, depending on query size, 3, 3, 2, or 1 filter banks for a total of 15, 9, 6, or 3 input streams.

Table 5-15 Per component resource utilization for the Altera Stratix-IV

Query Size Replications 2-Hit Streams	Logic Utilization			
	Component	ALM	M9K	M144K
Qs<256 Reps=5 2hSt=25	5 two hit	5771	250/100	0/9
	1 EUA (16 Folded)	4084	8	0
	1 SW (16 Folded)	1475	8/216	12/0
	Total	11363	266	12
Qs<512 Reps=5 2hSt=15	3 two hit	3544	200/60	0/10
	1 EUA (16 Folded)	6888	16	0
	1 SW (8 Folded)	5296	32/240	12/0
	Total	15750	248	12
Qs<1024 Reps=4 2hSt=12	3 two hit	3625	258/72	0/12
	1 EUA (16 Folded)	12540	32	0
	1 SW (8 Folded)	10473	64/272	12/0
	Total	26660	354	12
Qs<1024 Reps=3 2hSt=9	3 two hit	3704	368/96	0/16
	1 EUA (16 Folded)	23998	64	0
	1 SW (8 Folded)	20412	128/336	12/0
	Total	48163	496	16
Total Available (Stratix IV)		325000	1610	60

Table 5-16 Per component resource utilization for the Xilinx VirtexVI

Query Size Replications 2-Hit Streams	Logic Utilization		
	Component	Slices	BRAMs/FIFOs
Qs<256 Reps=3 2hSt=15	5 two hit	3921	159
	1 EUA (16 Folded)	2103	8
	1 SW (16 Folded)	1227	56
	Total	7313	223
Qs<512 Reps=3 2hSt=9	3 two hit	2496	128
	1 EUA (16 Folded)	3590	16
	1 SW (8 Folded)	4595	80
	Total	10725	224
Qs<1024 Reps=2 2hSt=6	3 two hit	2534	134
	1 EUA (16 Folded)	6304	32
	1 SW (8 Folded)	8804	112
	Total	17687	278
Qs<1024 Reps=1 2hSt=9	3 two hit	2537	140
	1 EUA (16 Folded)	12064	64
	1 SW (8 Folded)	17152	176
	Total	31801	380
Total Available (Stratix IV)		18560	720

For the following tests, we use a protein database with 15.4 M sequences and 5.4 G characters. The reference tests are run on the Convey machine's CPU which is a four-core Intel CPU (Xeon L5408 2.13 GHz). We chose this reference processor because it is of the same technical generation as the FPGAs in our system. All the reference and accelerated tests were done with the latest NCBI BLAST with the -num threads 4 option; this forces maximum useful parallelism for both the reference code and the CPU part of the accelerated code. In NCBI BLAST, the traceback code that generates the actual alignments is not threaded and, therefore, is completely serial in both reference and accelerated tests.

NCBI BLAST provides a wide range of user options that vary such quantities as internal thresholds and the quantity of results provided. The internal thresholds control sensitivity and, thus, the amount of work to be done. Varying them has comparable effect on both reference and accelerated execution. Accelerated BLAST and NCBI BLAST are not identical; however, accelerated BLAST executes exhaustive ungapped and gapped alignments, whereas NCBI BLAST executes gapped and ungapped extensions with complex heuristics. In order to guarantee no false negatives, it may therefore be necessary to increase the sensitivity (i.e., lower the threshold) in the accelerated BLAST. Note that, as long, as all false negatives are eliminated this does not change the overall output, and the final run of NCBI BLAST still uses the user specified thresholds and eliminates false positives. In contrast to the sensitivity parameters, those for output affect primarily the CPU-only part of the accelerated code. The default is to return the top 500 sequences of any possible statistical significance.

Given that the traceback code is serial (and given Amdahl's Law), permissive output has a disproportionate detrimental effect on the performance of accelerated BLAST.

We run four tests varying the following parameters: ungapped alignment threshold, gapped alignment threshold, E-value, and number of match sequences returned. Results are summarized in Table 5-18 .

Table 5-17 Percentage of Sequences Remaining After EUA and Smith Waterman Filters

Test	DB' Reduction%	DB" Reduction%
1	0.13	0.035
2	2.15	0.034
3	0.02	0.014
4	1.28	0.034

Table 5-18 Various tests of reference and accelerated BLAST for queries up to 256 characters.

Test	Ref. Time	1 FPGA Filter Only	4 FPGAs Filter Only	Post-Filter Search	Post-Filter Traceback	1 FPGA Total	4 FPGAs Total	1 FPGA Speedup	4 FPGAs Speedup	Acc %
1	46.5	7.1	1.9	1.5	1.9	10.5	5.3	4.4x	8.8x	98.4%
2	45.6	10.5	2.9	1.5	1.7	13.7	6.1	3.2x	7.5x	100%
3	48.9	7.3	2.0	1.2	1.3	9.8	4.5	5.0x	10.9x	96.4%
4	47.0	8.1	2.2	1.4	0.4	9.9	4.0	4.7x	11.7x	100%

Table 5-19 Tests 2 and 4 (see text) of reference and accelerated BLAST for all queries.

Test	Ref Time	1 FPGA Filter Only	4 FPGAs Filter Only	Post-Filter Search	Post-Filter Traceback	1 FPGA Total	4 FPGAs Total	1 FPGA Speedup	4 FPGAs Speedup	Acc %
2	78.5	12.6	3.4	2.4	0.99	16.0	6.8	4.9x	11.5x	99.99
4	68.2	11.2	3.0	2.2	0.80	14.2	6.0	4.8x	11.4x	100

First, we note the general effectiveness of the filtering mechanisms: depending on thresholds, the EUA filter reduces the original database from 98% to 99.98%, whereas the S-W filter reduces it by from 99.97% to 99.99%.

In Test 1. Reference and Accelerated NCBI BLAST have default parameters (E-value = 10, max target sequences = 500). The EUA threshold is set to the ungapped extension threshold of NCBI BLAST, whereas the S-W threshold is set to the gapped extension threshold. In this baseline test, we note that there are some false negatives, although none ever appear in the top 100 of returned sequences.

In Test 2. All parameters are again set to default, but for accelerated BLAST, the EUA threshold is reduced by 12. This selection is based on the analysis of the EUA and S-W scores of the missing sequences in comparison to their corresponding threshold. Because the S-W threshold is not changed, the reduced databases sizes (DB") are not significantly changed either. As a result, the postfilter timing remains the same. Reducing the EUA threshold increases the FPGA streaming time slightly. However, the accuracy is improved to 100% (no misses) but with a reduction in performance.

In Test 3. The E-value is reduced from the default value of 10 to 1.0 E-5 such that the returned sequences are more statistically meaningful. An E-value of 10 is considered too permissive for these sizes of databases. This test assesses the effect of the E-value on the performance and accuracy. As in Test 1, the thresholds used by FPGAs are those calculated by NCBI BLAST during ungapped and gapped extension. The reduction in E-values has little effect on the FPGA streaming time. However, the post-FPGA processing time is reduced producing slightly better speedup. The number of false negatives, however, increases to a greater value than the original.

In Test 4. The EUA threshold is reduced by 20%. Also, both reference and accelerated BLAST are tested with max target sequences of 50, which forces the tool to report the top 50 sequences only. The selection of 20% reduction as the EUA threshold, as

opposed to a constant reduction of 12 (as in Test 2) is based on the analysis of the scores of the missing sequences in Test 3. Because in Test 3 we used a more restrictive E-value, the default thresholds increased. The comparison of the scores of the missing sequences and the default thresholds in the other tests shows that with a 20% reduction in EUA threshold we can achieve 100% agreement.

Overall, this use case shows optimal performance and accuracy results. Table 5-19 shows results from Tests 2 and 4 done for a general set of 600 queries selected randomly from NR. We note that the end-to-end speedup of accelerated BLAST is around 5x when using a single FPGA and over 11x when using 4 FPGAs.

5.8 Summary

In this chapter, NCBI BLAST, the de facto standard for biosequence analysis is accelerated based on a novel pre-filtering approach. The prefiltering technique reduces the database size to a fraction of the original using three filters that emulate the three main phases of NCBI BLAST. The filters are either identical representations of the original or strictly more sensitive than the reference: that is, they might return more hits but they do not miss any hits that the reference might return.

For the word matching phase we used a two hit filter which returns a bit victor indicating exactly which diagonals have the two-hit property. Our two-hit filter is compact and accurate. The generated bit vector is used by the next filter: the Exhaustive Ungapped Alignment filter. The EUA filer emulates the ungapped extension phase of NCBI BLAST. Based on two novel techniques we effectively coupled the Two-Hit filter's bit vector with the EUA filter, such that the augmented EUA unit does not need prohibitive control logic

in order to skip the unpromising diagonals. Based on these optimizations our EUA filter is capable of consuming 3 to 5 residues per clock cycle depending on the query size. We load balanced system by replicating some number of two-hit filters per EUA filter. Using a single Virtex-6 FPGA, our pipelined system achieved 4x to 5x speedup over a four threaded CPU code without losing any sensitivity.

6 CLUSTALW

6.1 Overview

Biologists use approximate string-matching for pair-wise sequence alignment (SA) to determine, for example, how a newly identified protein is related to those previously analyzed and how it has diverged through mutation. Multiple-sequence alignment (MSA) extends this idea to more than two sequences: gaps are inserted as necessary to define a mapping of the sequences to rows of a matrix such that all columns have at least one letter.

MSA is the critical tool for extracting and representing biologically important, yet (potentially) faint or widely dispersed, commonalities from a set of strings [Gus97]. While SA is used to assign possible functions to a protein, MSA goes to the next level. Among its uses are prediction of function and secondary (two- and three-dimensional) structure, identification of the residues important for specificity of function, creation of alignments of distantly related sequences, and revealing clues about evolutionary history [Bar01].

While SA is typically used in database search (finding correlations of one sequence with millions of anonymous candidates), MSA is generally applied to some number of sequences that are already hypothesized to have some commonality, and, though it is often the case that some sequences are better understood or more important than others, MSA is basically an all-to-all matching problem. Another difference is that, whereas there is a consensus on the evaluation of SA alignments on the basis of Karlin-

Altschul statistics, with MSA there is no objective way to define an unambiguously correct alignment [Dur98].

These last facts have the following consequence. Evaluating MSA applications requires either expert knowledge or its surrogate through preselected sets of related sequences (e.g., BAliBASE [Bah01]) and encoded evaluation metrics (e.g., MetAl [Bla11] or BaliScore [Bah01]). In an MSA workflow, a number of sequences k of length n are aligned. The median value for n is about 300 but is often closer to 1,000, whereas k can range from a few to a few thousand sequences or more.

Optimal MSA algorithms have been created by extending DP-based SA to higher dimensions. These have exponential complexity in the number of sequences $O(n^k)$. Applying restrictions (see e.g., [Ben12b], [Car88]) results in tremendous speedups making them plausible for k up to small double digits. A larger k, however, requires the use of heuristics; these are generally a version of *progressive refinement* [Fen87]. These codes typically run in three phases: (i) an all-to-all phase where all pairs are aligned and scored, (ii) a tree-building phase where a guide tree is built that has sequences as its leaves and whose interior nodes represent alignments, and (iii) a final phase where all pairs of nodes are aligned.

The most commonly used MSA code is CLUSTALW [Tho94], but, although it is exponentially more efficient than the optimal methods, it still takes hours to days of CPU time for larger runs. Given that MSA is often a subroutine of a more complex task, such as finding evolutionary relationships, its acceleration is critical.

We now review some of the previous work. The first phase of CLUSTALW consists of over 90% of the execution time. It has been accelerated both with FPGAs [Oli05] and GPUs [Lip88]. Both of these studies follow the serial code in using dynamic programming (DP) for the all-pairs alignments and report factors of 40× to 50× speedups over a single core, respectively. We find that when the FPGA-DP method is ported to updated FPGAs and multicore CPUs, the speedup is in a similar range, but with some variance; i.e., from 18× to 58×. Lloyd and Snell have accelerated a generic third phase, which, for CLUSTALW, takes most of the remaining time on FPGAs, and obtained a speedup of up to 150× versus a single core [Lio11].

We use a different approach in creating a CLUSTALW-based, FPGA-accelerated MSA (FMSA). Just as BLAST applies multiple passes of heuristics to emulate DP-based SA, so we apply BLAST-inspired filters to the pair-wise alignments. In particular, we use a two-hit filter (seeding pass) [Jac08] followed directly in a pipeline by an ungapped alignment (ungapped extension pass) [Mah10],[Her07]. For the latter, we emulate the ungapped mode of NCBI BLASTp [Mah12a].

There are two versions of FMSA: fast (FMSAf) and emulation (FMSAe).v In both cases, we use a scoring function analogous to that used by CLUSTALW; i.e., rather than returning an E-value, FMSA computes a function based on identity counts. In fast mode, these scores are sent directly to the second phase of CLUSTALW to complete the processing. In emulation mode, some fraction of the high-scoring pairs are rescored using the DP-based method of Oliver et al. [Oli06]that emulates the CLUSTALW scoring function precisely. The result is a factor of from 80× to 189× speedup with respect to

eight-way parallel CPU code with the lower number corresponding to achieving results with quality comparable to the original.

The primary contribution of the work in this Chapter is an FPGA-accelerated version of ClustalW that achieves both substantial speedup over previous methods for computationally intensive data sets and high quality results that, especially in emulation mode, closely agree with those generated by the original code. The mechanism is the primary intellectual contribution of this Chapter and has three parts: (i) the overall approach where we apply prefiltering based on ungapped alignment and rescore as necessary, (ii) the modification of the original components to support an MSA rather than an SA scoring function, and (iii) the redesign of the filter sequence into a pipeline to avoid costly system overhead and reconfiguration. The significance is that—when coupled with the work of Oliver, et al. [Oli06], and of Lloyd and Snell [Lio11]—this could become the FPGA-accelerated MSA method of choice. We have developed FMSA using a standard high-end PC with a Gidel PROCe III accelerator board.

The rest of this chapter is organized as follows. We begin with a brief review of progressive alignment for MSA and ClustalW. Section 6.3 details our FPGA based ClustalW and Section 6.4 describes the results.

6.2 BACKGROUND

6.2.1 Basics of MSA for Biological Sequences

There are a number of heuristic MSA codes that use progressive sequence alignment. They differ in three ways: (i) the order of alignments, (ii) whether there is a single

growing alignment or multiple subfamilies that are later aligned to each other, and (iii) the procedure used to align score sequences and alignments against an existing sequence or alignment. Generally, a binary tree is constructed to guide the order of alignments with the most similar pairs--being the most reliable--being aligned first.

Scoring MSAs is an active area of research, but a commonly used metric is the sum-of-pairs (SP) score. This is a direct extension of pair-wise scoring. We follow the discussion in [Gus97]: given a multiple alignment *M* of *k* strings, an induced pairwise alignment between strings *Si* and *Sj* is obtained by removing all rows except for the *i*th and *j*th. The pair-wise score can be calculated using a standard SA function, or one selected for MSA. The SP score is the sum of all of the pair-wise scores.

In order to test the quality of an MSA algorithm, a preferred method is to evaluate it with a golden or reference data set; i.e., an alignment that has been created by a domain expert to deal with a specific, realistic, biological scenario. The BAliBASE 2.1 benchmark alignment database contains a number of case studies giving both sequences and a putative ideal reference alignment. Quality (determined by running the program BAliScore) is based on the SP score and computed as follows. For all columns in the test alignment and for each pair of residues, a score of 1 is given if the residues are aligned with each other in the reference alignment, and a 0 is given otherwise. This sum is normalized by the scores computed for the reference alignment.

A recent paper describes another set of distance measures for MSAs as follows :

hd: Simple homology distance gives the possibility that a randomly selected base x from an MSA will be aligned to a different location against a sequence randomly selected from the remaining sequences that do not contain x.

ssp: Symmetrized SP score is a symmetrized version of the SP score defined above.

pi: Positional information is incorporated into hd where gaps occur.

6.2.2 CLUSTALW Overview

From Table 6-1, we see that most of the effort is in phase 1; that is what we accelerate here. CLUSTALW improves the original progressive alignment methods by adding a number of heuristics. Most of these are incorporated into the second and third phases; and so need not be described here.

Table 6-1 PROFILE OF CLUSTALW BASE CODE W.R.T. VARIOUS DATA SETS

Benchmark	# of Seq	Phase 1	Phase 2	Phase 3
BB: MYB	180	88.0%	0.3%	11.7%
BB: 7tm	128	90.4%	0.04%	9.5%
NCBI 1	231	95.4%	0.2%	4.4%
NCBI 2	1000	91.4%	0.4%	8.0%
Average	--	91.3%	0.2%	8.4%

The first phase creates a matrix of alignment scores for all sequence pairs. Note that the CLUSTALW code appears to have been modified since the original paper; this is the code to which we refer. The scoring itself requires multiple passes. In the first two, a best local alignment is found with the use of a variation of the Myers and Miller algorithm, which uses a variation of global alignment with dynamic programming.

In the third pass, the actual score is determined from a count of the number of identical residue pairs in the optimal alignment. Then, this number is divided by the minimum of the lengths of the two sequences to create a similarity measure. Next, the result is subtracted from one to get the distance measure between the two sequences. One important point is that Myers and Miller is a memory efficient recursive global alignment algorithm: it divides the alignment space in half by dividing one sequence in half. It then finds the optimal point on the other sequence such that the concatenation of the two alignments of the subsequences on either side of the midpoint maximizes the global score. In this process it properly handles possible gaps in the neighborhood of the optimal midpoint.

We give this detail to show the challenge in exactly emulating the CLUSTALW scoring function. Besides the complexity, there can be multiple optimal alignments or traces between sequences; choosing the wrong one will lead to disagreement with the reference code. Oliver, et al., in their acceleration of the pairwise alignment phase [Oli06] (with a method based on S-W), have achieved near perfect agreement.

6.3 DESIGN AND IMPLEMENTATION

6.3.1 Design Overview

In all-pairs alignment, FMSA constructs a database of the sequences to be multiply aligned and consecutively matches sequences against the remainder. Although DP-based methods have excellent performance with respect to software, heuristic methods

are much faster still, in the same way that BLAST is substantially faster than S-W. FMSA uses two FPGA-based filters we have used previously to accelerate BLAST:

• The two-hit filter

• The exhaustive ungapped alignment (EUA) filter

For FMSAe we add an additional FPGA pass. The idea here is to expend marginal additional effort to improve agreement with the original code. For each sequence we rescore the highest scoring 10% of the sequences with the DP-based scoring function [Oli06] that nearly perfectly emulates the CLUSTALW scoring function.

6.3.2 FMSA Scoring

Before describing the details of the FMSA filters, we present some results of various possible scoring functions. We begin by introducing some terminology. *Si* is an input string, *LGA(Si* and *Sj)* represents the optimal local gapped alignment between *Si* and *Sj Score(Alignmenti)* returns the raw score of an *Alignmenti*, *NID(Alignmenti)* returns the number of identical residue pairs in an *Alignment i*, *LUA(Si, Sj, k)* represent the *kth* best local ungapped alignment between *S*i and *Sj* on the basis the raw scores. For each pair of sequences *Si* and *Sj*, CLUSTALW calculates the distance as follows:

$$dist_{ref}(Si, Sj) = 1 - \frac{NID(LGA(Si, Sj))}{\min(len(Si), len(Sj))}$$

Because our proposed method involves rescoring top pairs to improve agreement, we are looking for the scoring function which returns, as much as possible, the same top

scores as CLUSTALW. We tested five method as described below. The first three try to approximate the best local gapped alignment as a collection of some number of top local ungapped alignments. The 4th and 5th methods try to find a correlation between the identity count and the raw score of the top scoring alignments.

1) Find the best ungapped local alignment and count the number of identities:

$$dist_1(Si, Sj) = 1 - \frac{NID(LUA(Si, Sj, 1))}{\min(len(Si), len(Sj))}$$

2) Same as method 1, except add the top two best local ungapped alignments:

$$dist_2(Si, Sj) = 1 - \frac{\sum_{k=1}^{2} NID(LUA(Si, Sj, k))}{\min(len(Si, Sj))}$$

3) Same as methods 1 and 2, except add the top five best local ungapped alignments:

$$dist_3(Si, Sj) = 1 - \frac{\sum_{k=1}^{5} NID(LUA(Si, Sj, k))}{\min(len(Si, Sj))}$$

This method is similar to the ungapped option in NCBI BLASTp.

4) Find the best local ungapped alignment and use the raw score:

$$dist_4(Si, Sj) = 1 - \frac{\sum_{k=1}^{5} score(LUA(Si, Sj, k))}{\min(len(Si, Sj))}$$

5) Find the best local gapped alignment and use the raw score (S-W):

$$dist_5(Si, Sj) = 1 - \frac{Score(LGA(Si, Sj))}{\min(len(Si, Sj))}$$

The evaluation of these scoring functions is shown in Figure 6-1.

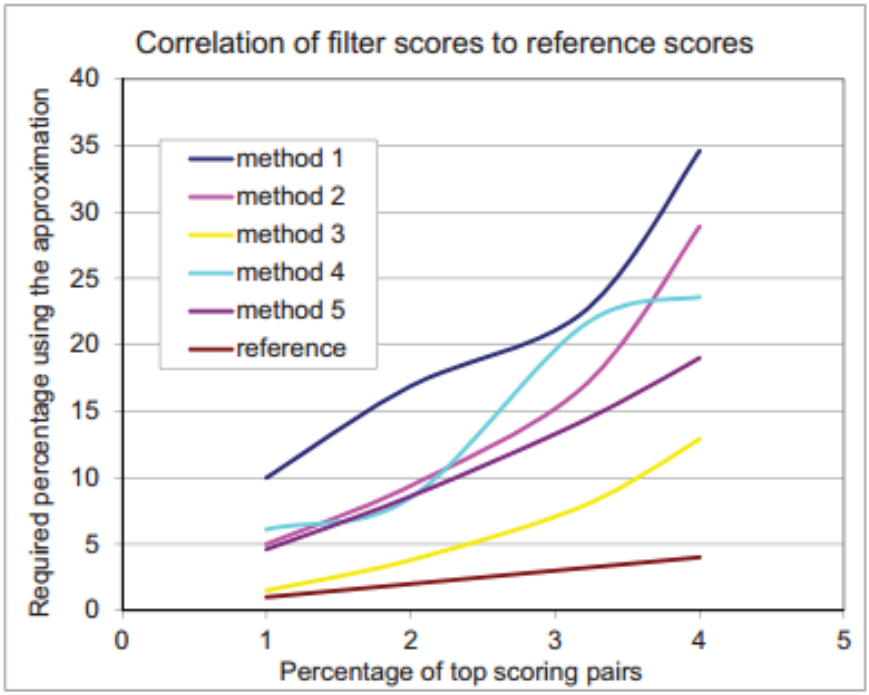


Figure 6-1 correlation of approximate filter scores to reference sores in clustal-w

The series shows the fraction of high-scoring pairs that must be rescored to guarantee that some top fraction of the high scoring pairs of CLUSTALW are matched. The lower the series the better. Note that, for method 3, rescoring 10% of the highest scoring pairs covers the top 3.6% from CLUSTALW. For method 1, nearly 30% must be rescored to achieve the same result.

Table 6-2 MEASURE OF THE BIAS AND STANDARD DEVIATION IN PAIRWISE SCORES BETWEEN ORIGINAL CLUSTALW AND FILTER OUTPUT

Database	# of Seq	Avg. of diffs	STD of diffs.
7tm	128	-0.01	0.03
Myb	180	-0.02	0.09
NCBI	231	-0.05	0.03

We selected method 3 for use by FMSAe. Table 6-2 shows the result of comparing all of the scores generated by FMSAe with CLUSTALW.

6.3.3 Filter Details

From the previous section we see that the scoring function requires finding the top ungapped alignments and then scanning them for identities. We use a two stage process. In the first, we follow standard BLAST procedure and eliminate all alignments where there are not any two seed matches within a certain distance (two-hit filter). In the second, we simultaneously perform two functions; i.e., we exhaustively scan all alignments for high-scoring ungapped local alignments and we count the identities in those alignments.

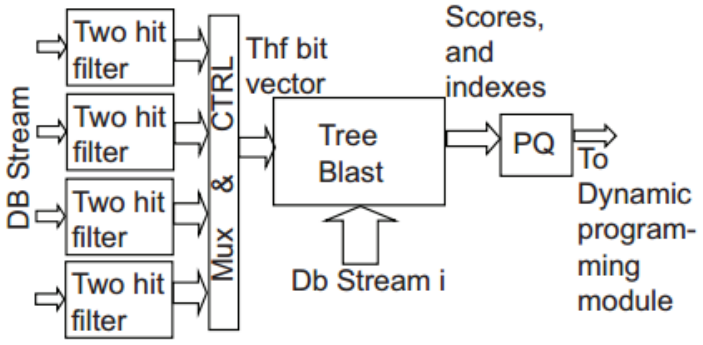


Figure 6-2 Filtering Pipeline for ClustalW

The designs of the two-hit filter and the exhaustive ungapped alignment filter are described in previous chapters, here we give a brief reminder. Figure 6-2 shows the overall scheme; i.e., parallel database streams feed the two-hit filters, which in turn feed a EUA filter. This structure is replicated some number of times depending on the sizes of the strings and of the FPGA. The EUA filter is capable of consuming three to five characters at each clock thanks to the two-hit filter data. The EUA reads in 16

characters and the corresponding filter bits as required. After processing of the data of one two-hit-filter, it starts working on the next sequence from the next two-hit filter.

In order to keep all of the units busy, the database sequences are sorted based on length and multiplexed among multiple two-hit filters. As a result the time required to process successive sequences is nearly equal. Note that there are two priority queues in the design. One priority queue is inside the EUA module and stores the top five local ungapped alignment scores. Another priority queue is outside EUA module to store the indexes of the top 10% scoring sequences. After EUA streams the entire database, the data in this second priority queue is passed to the dynamic programming module to perform the refinement.

6.4 RESULTS

We have implemented FMSA system on our Gidel Proc III board which is described in Section 2.6.2. For the Stratix III, for most problem sizes, we are able to map 16 two-hit filter units and 4 EUA units in a pipeline as described. For small problems with a maximum sequence size of less than 256, we can map eight replications (32 two-hit filters); if the sequence is larger than 1,024 we can map two replications (8 two-hit filters). The maximum sequence size in the database is used to pick the proper programming file to load to the FPGA. With 1 GB/s DMA capacity on the board, the transfer of sequence neighborhoods from host to the device memory takes a negligible fraction of a second. Much more bandwidth is available in current FPGA-based systems.

On the device, with two memory modules, each with nearly 4 GB/s bandwidth, 16 two-hit filters and 4 TreeBLAST modules easily work in parallel without hitting the bandwidth barrier. With the current working frequency of 140 Mhz, the required bandwidth adds up to 32 x 140 M = 4.5 GBs. For the emulation mode, each DP processing element requires 160 ALMs and one memory BRAM (M9k). On the Stratix III, if the maximum sequence size is less than 256, we can map two replications of the systolic array to FPGA, whereas, for 256-512 (average case) we can fit one instance. For larger sequences, folding is necessary. When folded n times, the streaming rate is reduced by a factor of n.

Table 6-3 QUALITY MEASURE OF FMSA-F AND ORIGINAL CLUSTALW WITH RESPECT TO THE BALIBASE BENCHMARK SUITE USING SP FROM THEBALISCORE CODE.

Database	# of Seq.	ClustalW	FMSA-F
7tm	128	0.822	0.747
Myb	180	0.969	0.850
Kinase3	19	0.777	0.827
Kinase2	18	0.739	0.738
1ajsa	28	0.405	0.464
1idy	27	0.591	0.554
1lvl	24	0.836	0.881
1aboA	5	0.688	0.558
1lcf	6	0.947	0.928

Recall that FMSA can run in two modes: fast and emulation. Table 6-3 shows a measure of quality of the FMSAf with respect to the BAliBASE benchmark and the SP metric. Although not conclusive, we note that the results are not unreasonable, even without rescoring.

Table 6-4 QUALITY MEASURE OF FMSA-F AND ORIGINAL CLUSTALW WITH RESPECT TO THE BALIBASE BENCHMARK SUITE USING SP FROM THE BALISCORE CODE.

Database	ClustalW vs. Reference	FMSA-f vs ClustalW	FMSA-f vs. Reference	FMSA-e vs. ClustalW	FMSA-e vs. Reference
Myb	0.97	0.84	0.85	0.98	0.99
7tm	0.82	0.78	0.76	0.88	0.80
NCBI	-	0.93	-	0.94	-

Table 6-5 QUALITY MEASURE OF FMSA-F, FMSA-E, AND ORIGINAL CLUSTALW WITH RESPECT TO THE BALIBASE BENCHMARK SUITE USING VARIOUSDISTANCE METRICS FROM METAL [Bla11]

Database	ClustalW v. Reference			FMSA-f v. Reference			FMSA-e v. Reference		
	Hd	Ssp	Pi	hd	ssp	Pi	hd	ssp	pi
Myb	0.28	0.55	0.26	0.38	0.61	0.33	0.27	0.54	0.24
7tm	0.30	0.38	0.24	0.37	0.47	0.31	0.32	0.41	0.26

More detailed quality results, albeit for fewer sequences, are given in Table 6-4 and Table 6-5. We use the two larger studies from BAliBASE plus a synthetic database generated from NCBI BLASTp where we simply scanned a random sequence and retained the top 231 scoring sequences from NR. In Table 6-4 we use the SP metric from BaliScore. We compare the original CLUSTALW code, FMSAf, and FMSAe to the reference MSA. We find that FMSAe has a nearly identical quality to CLUSTALW, but FMSAf also shows a high degree of agreement. We also compare FMSA with CLUSTALW. For FMSAe, we find a high correlation; not surprisingly, FMSAf is not as correlated but still has a high correlation. In Table 6-5 , we show results with respect to the MetAl distance metrics. Again, we compare the original CLUSTALW code, FMSAf, and FMSAe to the reference MSA. For FMSAe, we again find that the distance from the reference MSA is nearly identical to that of CLUSTALW, FMSAf lagging somewhat but still clearly in the same range.

Table 6-6 PERFORMANCE FOR MSA RUNS OF 1000 SEQUENCES. ALL TIMES IN SECONDS. THE 8 CORE PC ASSUMES IDEAL PARALLELIZATION. DP, FMSA-F,AND FMSA-E COLUMNS GIVE BOTH TIME AND SPEED-UP WITH RESPECT TO 8 CORE PC.

Max Seq. Len.	Overhead	FPGA filter	FPGA rescore	PC(8 core)	DP	FMSA-E	FMSA-F
256	0.9	0.2	0.3	150	3.5/43x	1.4/107x	1.1/136x
512	1.6	0.7	0.7	434	7.5/58x	3.0/145x	2.3/189x
1024	2.4	1.2	3.0	549	31/18x	6.6/83x	3.6/152x

Table 6-6 shows performance of the original CLUSTALW, its DP-based acceleration, and the acceleration with FMSAf and FMSAe on the reference system. For the CPU-only version, we simply assume the best case of perfect eight-way threading. This appears to be about a factor of two more than has been achieved so far (see [Lio11] for a discussion), but appears to be plausible. The performance of FMSAf is that of FMSAe minus the FPGA rescore time and a small amount of the overhead. We note that FMSA is from 83× to 189× faster than the CPU version and from 2.5× to 8.4× faster than the DP-based method. The greater advantage is for the larger problem size.

6.5 Summary

In this chapter we described an FPGA-accelerated MSA program based on ClustalW. It differs from previous accelerations in that it uses BLAST heuristics rather than dynamic programming. We used our Two-Hit filter and EUA filter to approximate the DP method. In order to do so we augmented the EUA filter to count the identities in addition to the raw score. We showed that a combination of the top local ungapped alignments have sufficient correlation with the best local gapped alignment.

Our system achieves many-fold speedup over the DP-based code, which itself has better performance than the CUDA version. We have created two versions, one that successfully emulates ClustalW, the other that gives results of somewhat lower quality, but with roughly twice the performance.

7 Conclusion and Future Work

We conclude this study by summarizing our work in acceleration of biosequence analysis tools. We present our reflections on using filtering for their acceleration. Finally, we will present some guidelines for future work.

7.1 Summary

In this work, we have studied and tested various acceleration engines for biosequence analysis tools. Our research includes FPGA-based acceleration, FPGA-based algorithm design, performance analysis, scalability analysis, system-level testing and verification, and algorithmic optimizations/approximations for hardware acceleration.

We implemented two acceleration engines for NCBI BLASTp. We conducted extensive system-level tests on two different acceleration systems. We were able to generate transparent results compared to production-level code. We also implemented and tested a production-level acceleration engine for a multiple sequence alignment tool. We demonstrated significant speedup over the original code with reasonable accuracy using a novel approximation method.

We learned the following lessons from our study:

Prefiltering is a tricky approach to accelerating database query and processing applications. A prefiltering approach can be defined as follows: a fast-filtering engine that is based on an approximation method is used to reduce the size of the database to a small fraction of the original. The filter should be more sensitive than the original code. While it can return more sequences than the reference code, it should return all the

sequences that the reference returns. There are several challenges with the application of prefiltering for acceleration purposes:

- **Postfilter overhead**: Often, a filter reduces the size of the database, but the actual results are produced after the reduced database is generated. The runtime of this postfiltering calculation should be a small fraction of the original, otherwise the filter will act as an overhead. This is especially true when the software uses seed-based heuristics (as is the case with most of the sequence analysis tools).
- **Accuracy and performance tradeoff**: Given that accuracy is a soft constraint, the programmer can trade off accuracy and performance. In such a case, it is possible to use approximation and even heuristics in order to simplify the hardware design and boost performance. Unfortunately, this is not always a given. On one hand, if the designer can find an approximation that is more sensitive than the original code, perfect accuracy can be achieved, and, assuming, the postfilter job can be efficiently performed on the host, the performance gain can be impressive.

Implementation of seeding heuristic in hardware requires very careful design. Throughout this study, we learned that the implementation and exploitation of a seeding heuristic in hardware can be both challenging and beneficial. The FPGA block RAMs provide a convenient parallel interface for accessing seeding indexes or profiles. On one hand, seeding indexes that are mapped to hardware block RAMs or lookup tables use precious resources. The designer should assess the pros and cons of the performance gain vs. the resource usage. The designer should also consider the overhead of combining the seeding output with the rest of the tool chain. Obviously,

adapting the phases after seeding in order to take advantage of the information should not impose massive overhead on the original filter.

Given that there is no overhead in reprogramming, a multiphase system can deliver the highest performance among all the acceleration engines of a multi-stage heuristic sequence analysis tool. The general rule of thumb in engineering is to keep the design simple. A multiphase system follows this same idea. Sequence analysis applications are often fully parallelizable. This applies to the substages of the application as well. Thus, the programmer can replicate each stage's hardware units maximally, run each stage in hardware, save the results, and start the next stage. The design benefits from the removal of glue logic and the overhead of stalling that are inevitable in any data-dependent execution an algorithm on a tool chain.

Any streaming memory interface has its own overhead. Memory modules provide the best throughput when data are read or written sequentially. Random memory access can be up to 16x more costly than sequential access. This encourages the designer to use streaming interface**.** Interestingly for sequence analysis, streaming looks ideal; each subject sequence is a character string that can be read sequentially. On the other hand, implementing a fully tested and reliable streaming interface can be challenging if several streams must be bundled. For example, we spent a great deal of time testing our multiphase system, which required the accurate alignment of the bit vector and the database stream in the EUA filter. The best solution is usually to combine a complete streaming interface with occasional random access in order to improve the reliability of the system.

Careful load balancing is required to gain performance in a pipelined system. Stalling is inevitable in a pipelined system with data-dependent heuristics. Thus careful load balancing based on statistical analysis and simulation is needed.

For applications like NCBI BLAST, which have multiple phases that are equally time consuming, FPGA acceleration is especially challenging. First one needs to note that the more complicated an application is, the harder it is to map it to an FPGA. BLAST is certainly a complicated application with many heuristics. Second, the three phases of BLASTp contribute equally to the running time. Third, high profile tools like BLAST are constantly being updated and improved. Finally, software acceleration solutions based on multithreading, SIMD extensions, and GPUs also offer significant speedups.

Approximation can help the FPGA designer if there is room for any divergence from the production code. For example, in acceleration of Clustal-W, we achieved an order of magnitude speed-up over an exact FPGA based solution. This was mainly due to the fact that we approximated the distance matrix using a more compact and highly optimized EUA engine. **An optimization that is only marginally useful for the CPU implementation turned out to be highly beneficial for the FPGA version.**

7.2 Future Directions

For further study, there are a number of directions that can be considered.

- **porting the tool chain system to Virtex-7**

The latest Convey machine uses high end Virtex-7 FPGAs. It would be interesting to port the tool chain system to this machine and to measure its performance. Our IO interface currently support up to 36 streams. Porting to Virtex-7 will require additional changes to the IO interface so that we can at lease support 72 streams.

- **Acceleration of other versions of NCBI BLAST**

NCBI BLAST has several versions. We have analyzed the most challenging version, NCBI BLASTp, which is used for protein sequence alignments. NCBI BLASTn, for example, is used to align genomic sequences. Only two bits are required to represent genomic residues, and alignment scoring is performed with a simple weighted edit distance. These are both ideal for FPGAs because they can be efficiently mapped to small modul-4 processing units.

- **Acceleration of next-generation sequencing tools**

Next-generation sequencing machines can generate billions of short or long reads in a very short amount of time. Compared to the traditional shotgun sequencing approach, they demand to a larger degree more throughput and computational capacity. It would be interesting to study the possibility of accelerating these tools based on our filtering and seeding approach.

8 References

[Ach07] Nair, Achuthsankar S. "Computational biology & bioinformatics: a gentle overview." *Communications of Computer Society of India* 2 (2007): 1-13.

[Alb02] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. Molecular biology of the cell. Hardcover, 2002.

[Alp95] Alpern, Bowen, Larry Carter, and Kang Su Gatlin. "Microparallelism and high-performance protein matching." *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM, 1995.

[Alt90] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. Basic local alignment search tool. Journal of Molecular Biology 215 (1990), 403–410.

[Awa09] Awad, M. "FPGA supercomputing platforms: a survey." *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009.

[Bah01] A. Bahr, J. Thompson, J.-C. Thierry, and O. Poch, “BAliBASE (Benchmark Alignment dataBASE): enhancements for repeats, transmemebrane sequences and circular permutations,” Nucleic Acids Research, vol. 29, no. 1, pp. 323–326, 2001.

[Bar01] G. Barton, “Creation and analysis of protein multiple sequence alignment,” in Bioinformatics: A Practical Guideto the Analysis of Genes and Proteins, A. Baxevanis and B. Ouellette, Eds. Wiley, 2001.

[Ben12a] Benkrid, Khaled, et al. "High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP." *International Journal of Reconfigurable Computing* 2012 (2012): 7.

[Ben12b] Benson, Dennis A., et al. "GenBank." *Nucleic Acids Research* 40.D1 (2012): D48-D53.

[Bil06] Y. Bilu, P. Agarwal, and R. Kolodny, “Faster algorithmsfor optimal multiple sequence alignment based on pairwise comparisons,” IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 3, no. 4, pp. 408–422, 2006.

[Bla11] B. Blackburne and S. Whelan, “Measuring the distance between multiple sequence alignments,” Bioinformatics, vol. Online preprint posted 12/23/2011, 2011.

[Bra01] R.C Braun, K.T Pedretti, T.L Casavant, T.E Scheetz, C.L Birkett, C.A Roberts, Parallelization of local BLAST service on workstation clusters, Future Generation Computer Systems, Volume 17, Issue 6, April 2001, Pages 745-754.

[Bak10] Bakos, J. High-Performance Heterogeneous Computing with the Convey HC-1. Computing in Science and Engineering 12, 6 (2010), 80–87.

[Car88] H. Carrillo and D. Lipman, “The multiple sequence alignment problem in biology,” SIAM Journal of Applied Math., vol. 48, no. 5, pp. 1073–1081, 1988.

[Che07] Chen, T., et al. QCDOC: Cell broadband engine architecture and its first implementationa performance view. IBM Journal of Research and Development, 51(5):559–572, 2007.

[Chi97] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. Technical Report TR97-005, University of Minnesota, Computer Science Department, 1997

[Chi02] Chi, E., Shoop, E., Carlis, J., Retzel, E., and Bjornson, R.D.; Sherman, A.H.; Weston, S.B.; Willard, N.; Wing, J.; "TurboBLAST : a parallel implementation of blast built on the turbohub," Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM , vol., no., pp.183-190, 2002.

[Cho91] Chow, E.; Hunkapiller, T.; Peterson, J.; Waterman, M.S., "Biological information signal processor," *Application Specific Array Processors, 1991. Proceedings of the International Conference on* , vol., no., pp.144,160, 2-4 Sep 1991.

[Con13a] Convey Computer Corporation. Convey HC-2 Architectural Overview. www.conveycomputer.com/ files/4113/ 5394/7097/Convey HC-2 Architectural Overview.pdf, 2013.

[Con13b] Convey Computer Corporation. Hybrid-Core Computing for High Throughput Bioinformatics. www.conveycomputer.com/ files/2613/ 5085/5888/ConveyBioinformatics web.pdf, 2013.

[Cou05] Coulouris, G. BLAST benchmarks. NCBI/NLM/NIH Presentation, June 2005.

[Cul97] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. 1997. *Parallel Computer Architecture: A Hardware/Software Approach* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Dar03] Darling, Aaron, Lucas Carey, and Wu-chun Feng. "The design, implementation, and evaluation of mpiBLAST." *Proceedings of ClusterWorld* 2003 (2003).

[Don12] J. J. Dongarra and A. J. van der Steen. High-performance computing systems: Status and outlook. Acta Numerica, 21:379–474, 2012.

[Dur98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, Biological sequence analysis. Cambridge, U.K.: Cambridge University Press, 1998.

[Gok05] M.B. Gokhale and P.S. Graham, Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays, Springer, 2005.

[Gus97] D. Gusfield, Algorithms on Strings, Trees, and Sequences:Computer Science and Computational Biology. Cambridge, U.K.: Cambridge U. Press, 1997.

[Edg04] Edgar RC. MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res. 2004;vol. 32. pp: 1792-1797.

[Eur07] Sotiriades, Euripides, and Apostolos Dollas. "A general reconfigurable architecture for the BLAST algorithm." *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 48.3 (2007): 189-208.

[Ewe05] Ewens, Warren J., and Gregory Robert Grant. *Statistical methods in bioinformatics: an introduction*. Springer New York, 2005.

[Far07] Farrar, Michael. "Striped Smith–Waterman speeds database searches six times over other SIMD implementations." *Bioinformatics* 23.2 (2007): 156-161.

[Fen87] D.F. Feng and R. Doolittle, “Progressive sequence alignment as a prerequisite to correct phylogenetic trees,” Journal of Molecular Evolution, vol. 25, pp. 351–360, 1987.

[Gar06] Gardner, N., Feng, W., Archuleta, J., Lin, H., and Ma, X. Parallel genomic sequence-searching on an adhoc grid: Experience, lessons learned, and implications. In Supercomputing (2006).

[Geo11] George, A.; Lam, H.; Stitt, G., "Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing," *Computing in Science & Engineering* , vol.13, no.1, pp.82,86, Jan.-Feb. 2011.

[Gid13] http://www.gidel.com/

[Gid10] GiDEL, Inc. PROC Boards. www.gidel.com, Accessed 1/2010.

[Gra02] J. Grant, R. Dunbrack Jr., F. Manion, and M. Ochs. BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18(5), 2002.

[Har07] Harris, Brandon, et al. "A banded Smith-Waterman FPGA accelerator for Mercury BLASTP." *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. IEEE, 2007.

[Hen92] Henikoff, Steven, and Jorja G. Henikoff. "Amino acid substitution matrices from protein blocks." *Proceedings of the National Academy of Sciences* 89.22 (1992): 10915-10919.

[Hen10] Li, Heng, and Nils Homer. "A survey of sequence alignment algorithms for next-generation sequencing." *Briefings in bioinformatics* 11.5 (2010): 473-483.

[Her07] Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. Single pass streaming BLAST on FPGAs. Parallel Computing 33, 10-11 (2007), 741–756.

[Hig98] Higgins DG, Sharp PM, “CLUSTAL: a package for performing multiple sequence alignment on a microcomputer”.Gene 73 (1): 237–244. 1998.

[Hom09] Homer, Nils, Barry Merriman, and Stanley F. Nelson. "BFAST: an alignment tool for large scale genome resequencing." *PLoS One* 4.11 (2009): e7767.

[Int11] http://software.intel.com/en-us/avx/

[Jac07] Jacob, Arpith, et al. "FPGA-accelerated seed generation in Mercury BLASTP." *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007.

[Jac08] Jacob, A., Lancaster, J., Buhler, J., Harris, B., and Chamberlain, R. Mercury BLASTP: Accelerating protein sequence alignment. ACM Transactions on Reconfigurable Technology and Systems 1, 2 (2008).

[Jon04] Jones, Neil C., and Pavel Pevzner. *An introduction to bioinformatics algorithms*. the MIT Press, 2004.

[Kat02] Katoh,K., Misawa,K., Kuma,K. and Miyata,T. (2002) MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. Nucleic Acids Res., 30, 3059–3066.

[Ken02] Kent, W. James. "BLAT—the BLAST-like alignment tool." *Genome research*12.4 (2002): 656-664.

[Kor03] Korf, Ian, Mark Yandell, and Joseph Bedell. *Blast*. O'Reilly Media, Inc., 2003.

[Kri07] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin,K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the mercury system. J. VLSI Signal Process. Syst., 49(1):101–121, 2007

[Kuo08] Kuon, Ian, Russell Tessier, and Jonathan Rose. "Fpga architecture: Survey and challenges." *Foundations and Trends® in Electronic Design Automation* 2.2 (2008): 135-253.

[Lav06] Lavenier, D., Xinchun, L., and Georges, G. Seed-based genomic sequence comparison using a FGPA/FLASH accelerator. In Proc. IEEE Conference on Field Programmable Technology (2006), pp. 41–48.

[Li03] Li, Kuo-Bin. "ClustalW-MPI: ClustalW analysis using distributed and parallel computing." *Bioinformatics* 19.12 (2003): 1585-1586.

[Lin08] Lindholm, Erik, et al. "NVIDIA Tesla: A unified graphics and computing architecture." *Micro, IEEE* 28.2 (2008): 39-55.

[Lin08] Lindholm, Erik, et al. "NVIDIA Tesla: A unified graphics and computing architecture." *Micro, IEEE* 28.2 (2008): 39-55.

[Lin10] Ling, C., and Benkrid, K. Design and implementation of a CUDA-compatible GPU-basedcore for gapped BLAST algorithm. Procedia Computer Science 1, 1 (2010).

[Lio11] Lloyd, Scott, and Quinn O. Snell. "Accelerated large-scale multiple sequence alignment." *BMC bioinformatics* 12.1 (2011): 466.

[Lip87] Lipton.R. and Lopresti,D. (1987) Comparing long strings on a short systolic array. In: Systolic Arrays, (Moore,W., McCabe.A. and Urquhart,R. eds) pp. 363-376. Adam Hilger Boston, MA

[Lip88] H. Carrillo and D. Lipman, “The multiple sequence alignment problem in biology,” SIAM Journal of Applied Math., vol. 48, no. 5, pp. 1073–1081, 1988.

[Liu09] Liu, Yongchao, Douglas L. Maskell, and Bertil Schmidt. "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units." *BMC research notes* 2.1 (2009): 73.

[Liu11] Liu, Weiguo, Bertil Schmidt, and Wolfgang Muller-Wittig. "CUDA-BLASTP: accelerating BLASTP on CUDA-enabled graphics hardware." *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 8.6 (2011): 1678-1684.

[Lop87] D. P. Lopresti, P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences, Computer, v.20 n.7, p.98-99, July 1987.

[Mah10] A. Mahram and M. Herbordt, “Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Pre-filtering,” in Proceedings of the 24th ACM International Conference on Supercomputing, 2010, pp. 73–82.

[Mah12a] Mahram, Atabak, and Martin C. Herbordt. "CAAD BLASTP 2.0: NCBI BLASTP accelerated with pipelined filters." *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012.

[Mah12b] Mahram, Atabak, and Martin C. Herbordt. "FMSA: FPGA-Accelerated ClustalW-Based Multiple Sequence Alignment through Pipelined Prefiltering."*Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012.

[Mat03] D. Mathog, "Parallel BLAST on Split Databases," *Bioinformatics,* vol. 19, pp. 1865-1866, 2003.

[Met09] Metzker, Michael L. "Sequencing technologies—the next generation." *Nature Reviews Genetics* 11.1 (2009): 31-46.

[McG04] McGinnis, S., and Madder, T. BLAST: at the core of a powerful and diverse set of sequence analysis tools. Nucleic Acids Research 32 (2004), Web Server Issue.

[Mit10] Mitrionics. Mitrion-Accelerated NCBI BLAST for SGI BLAST. Available at www.mitrionics.se/press, Accessed 1/2010.

[Mur05] Muriki, K., Underwood, K., and Sass, R. RC-BLAST: Towards an open source hardware implementation. In Proc. International Work. High Performance Computational Biology (2005).

[Mur05] Muriki, Krishna, Keith D. Underwood, and Ron Sass. "RC-BLAST: Towards a portable, cost-effective open source hardware implementation." *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005.

[NCBa]http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt

[Nee70] Needleman, Saul B., and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins."*Journal of molecular biology* 48.3 (1970): 443-453.

[Nic10] Nickolls, J.; Dally, W.J., "The GPU Computing Era," *Micro, IEEE* , vol.30, no.2, pp.56,69, March-April 2010.

[Not00] Notredame C, Higgins DG, Heringa J. "T-Coffee: A novel method for fast and accurate multiple sequence alignment". J Mol Biol 302 (1): 205–217, 2000.

[Oli04] Oliver, Timothy, and Bertil Schmidt. "High performance biosequence database scanning on reconfigurable platforms." *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004.

[Oli05] Oliver, Tim, et al. "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW." *Bioinformatics* 21.16 (2005): 3431-3432.

[Oli06] T. Oliver, B. Schmidt, Y. Jakop, and D. Maskell, “Acceleratingthe viterbi algorithm for profile hidden markov models using reconfigurable hardware,” in LNCS v3991, 2006.

[Olu05] Olukotun, Kunle, and Lance Hammond. "The future of microprocessors." *Queue*3.7 (2005): 26-29.

[Pan11] Vouzis, Panagiotis D., and Nikolaos V. Sahinidis. "GPU-BLAST: using graphics processors to accelerate protein sequence alignment." *Bioinformatics*27.2 (2011): 182-188.

[Par09] Park, Jin H., Yunfei Qiu, and Martin C. Herbordt. "CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Accelerated Pre-Filtering." *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009.

[Pat90] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Ran05] Rangwala, H., Lantz, E., Musselman, R., Pinnow, K.,Smith, B., and Wallenfelt, B. Massively parallel BLAST for the Blue Gene/L. In Proc. HighAvailability and Performance Computing Workshop (2005).

[Rog00] Rognes, Torbjørn, and Erling Seeberg. "Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors." *Bioinformatics* 16.8 (2000): 699-706.

[Rog11] Rognes, Torbjørn. "Faster Smith-Waterman database searches with inter-sequence SIMD parallelization." *BMC bioinformatics* 12.1 (2011): 221.

[Sco10] Hauck, Scott, and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010.

[She08] Shendure, Jay, and Hanlee Ji. "Next-generation DNA sequencing." *Nature biotechnology* 26.10 (2008): 1135-1145.

[Smi81] Smith TF, Waterman MS: Identification of common molecular subsequences. J *Mol Biol* 1981, 147(1)**:**195-197

[Sza08] Szalkowski, Adam, et al. "SWPS3–fast multi-threaded vectorized Smith-Waterman for IBM Cell/BE and× 86/SSE2." *BMC Research Notes* 1.1 (2008): 107.

[Tim10] Time Logic Corp. Web Site. www.timelogic.com, Accessed 1/2010.

[Tho94] J. Thompson, D. Higgins, and T. Gibson, “CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,” Nucleic Acids Research, vol. 22, no. 22, pp. 4673–4680, 1994.

[Top13] http://www.top500.org/

[Wir08] Wirawan, Adrianto, et al. "CBESW: sequence alignment on the playstation 3." *BMC bioinformatics* 9.1 (2008): 377.

[Woz97] Wozniak, Andrzej. "Using video-oriented instructions to speed up sequence comparison." *Computer applications in the biosciences: CABIOS* 13.2 (1997): 145-150.

[Xil13] http://www.xilinx.com/

[YSM09] Liu, Yongchao, Bertil Schmidt, and Douglas L. Maskell. "MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA." *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009.

[Zha07] Zhang, Peiheng, Guangming Tan, and Guang R. Gao. "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform." *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*. ACM, 2007.

[Zou12] Zou, Dan, Yong Dou, and Fei Xia. "Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA."*Concurrency and Computation: Practice and Experience* 24.14 (2012): 1625-1644.