DESIGN AND ANALYSIS OF BIOINFOMATICS ALGORITHMS ON AN FPGA PLATFORM

CHEN YUPENG

SCHOOL OF COMPUTER ENGINEERING

A thesis submitted to the Nanyang Technological University

in partial fulfillment of the requirement for the degree of

Doctor of Philosophy

2014

Acknowledgement

As a PhD student at Nanyang Technological University (NTU), I have got a great opportunity to access the most advanced technologies in a wide range of research areas. NTU provides me the chance to work with top researchers from all over the world. The well-equipped laboratory provides a very good environment to conduct my research work. All these experience is a great fortune and it will help me to continue my research in the future.

I feel so lucky that I have two great supervisors: Dr. Douglas L. Maskell and Dr. Bertil Schmidt. I would like to express my deepest gratitude and respect for their tremendous patience, guidance, support, and insights over past five years.

I would like to thank my friends Cui Jin and Liu Yongchao for the incisive discussions and their constructive suggestions. I would like to thank all other members in CHiPES for their help. I really have a great time in CHiPES.

Finally, I would like to express my special thanks to my parents and my girl friend for their love and support.

Table of Contents

Abstract

The appearance of newer sequencing technologies has largely improved the development of molecular biology and genomic research. A large volume of gene information or protein data can be generated with lower cost, which leads to the exponential growth of existing gene banks or databases. Thus, it becomes a challenging task for conventional algorithms or tools to extract information with biological significance among these ever increasing databases. There is an urgent need for innovative methods, algorithms, or tools to accomplish these complicated data analysis tasks on a more computationally powerful platform. After decades of development, the FPGA has proved itself in the field of high performance reconfigurable computation. For each generation, one can expect an immediate performance boost with the help of newer manufacturing technologies and a larger volume of resources on a single chip, both of which make it a competitive candidate for application acceleration.

This thesis investigates the potential of solving bioinformatics problems utilizing the outstanding computation capability of FPGAs. To give a quantitative analysis on an FPGA's performance, we choose two bioinformatics problems as the case study: genome search and short read alignment. An efficient architecture is proposed and implemented to maximize the performance of each application on a Virtex-5 FPGA platform.

For the genome search problem, we focused on the most time-consuming stage of NCBI BLASTN. Two innovative data structures, a parallel partitioned Bloom filter and a block hash table, are proposed to improve the computation efficiency. The Bloom filter design can support 16 data queries simultaneously with high filtration efficiency; while the latter bypasses the stringent requirement of a perfect/near-perfect hash function. The final system achieves over 10 times speedup against the single-thread NCBI BLASTN program testing with a 100kbase query sequence. In

addition, the experiments also show that the match rate together with the degree of parallelism influence the system's overall performance.

For the short read alignment problem, the key is to efficiently locate the segment with a high degree of similarity in the reference genome for each short read. We choose the banded Smith-Waterman algorithm as the align core to narrow down the search space. An innovative architecture is proposed to approximate the conventional algorithm but with a higher degree of parallelism. The align core alone achieves over 30 times speedup. The final design is a hybrid system utilizing both CPU and FPGA. The influence of different partitioning strategies is examined and the performance analysis also indicates that the performance bottleneck changes from the alignment computation to the candidate region search.

Publications

Journal Papers

1. Yupeng Chen, Bertil Schmidt, Douglas L. Maskell, “Reconfigurable accelerator for the word-matching stage of BLASTN,” *IEEE Transactions on VLSI Systems*, 21(4):659-669 (2013)
2. Yupeng Chen, Bertil Schmidt, Douglas L. Maskell, "A hybrid short read mapping accelerator," *BMC Bioinformatics*, 14:67 (2013)

Conference Papers

1. Yupeng Chen, Bertil Schmidt, Douglas L. Maskell, "An FPGA aligner for short read mapping," *International Conference on Field Programmable Logic and Applications* (FPL) 2012: 511 – 514
2. Yupeng Chen, Bertil Schmidt, Douglas L. Maskell, "Accelerating short read mapping on an FPGA (abstract only)," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (FPGA) 2012: 265
3. Yupeng Chen, Bertil Schmidt, Douglas L. Maskell, "A reconfigurable Bloom filter architecture for BLASTN," *International Conference on Architecture of Computing Systems* (ARCS) 2009: 40 – 49

Chapter1

Introduction

1.1 Overview

Computational Biology and Bioinformatics are fields involving analysis of a large amount of information of biological origins. Many research areas are involved in this interdisciplinary field, such as, sequence analysis, genome annotation, computational evolutionary biology, and so on. Taking sequence analysis as an example, many aspects of significant biological importance (such as RNA genes, genes that encode proteins, structural motifs, regulatory sequences, and repetitive sequences) can be studied by analyzing the sequence information. A genome comparison within a species can disclose similarity between protein functions, while the genome comparison among different species can be used to construct the phylogenetic tree. Computer programs are used daily to analyze sequences from more than 260,000 organisms, containing over 190 billion nucleotides [14] and the databases containing the biological information keep growing rapidly (they double their sizes approximately every 12-16 months). Figure 1.1 shows the growth of the GenBank DNA database and the UniProtKB/TrEMBL databases in terms of the number of base pairs (bps). The rapid growth of these databases poses an urgent need to accelerate the computational analyses in a corresponding manner. Another challenge in this field is that the analysis or computation process is very time consuming and sometimes it also requires intensive memory accesses. For example, DNA sequence analysis attempts to identify similar regions between two sequences from a biological point of view, i.e. maximize the number of identical base pairs while keeping the number of differences to a minimum, which requires complex computations. Another example with intensive computation is shotgun sequencing. Thousands of small overlapping DNA fragments (ranging from 35bp to 900bp) are generated which then need to be aligned to form a complete genome. The assembly

of these fragments is a complicated process, especially when the genome size is large. High-speed super computers can be applied to accelerate these analysis processes, but these machines are generally very expensive and limited in number. Their accessibility is still limited to small groups of researchers. A tool able to run on a lower cost general computation platform, such as a workstation or desktop PC, would be more attractive for molecular biologists.

In general, three approaches exist to accelerate the computations involving this large volume of data. The first approach is to design parallel algorithms running on general-purpose multi-core processor based PCs. Multi-core central processing units (CPU) can achieve a speedup factor proportion to the number of cores. Amdahl's law [9] gives a speedup estimation based on the number of cores and the fraction of computation that can be parallelized. If the cache within each core is optimally utilized rather than using the much slower main memory, an even better performance can be expected. The difficulty with this approach lies in how to parallelize the algorithm or the data flow to the different cores efficiently. The second approach is to utilize specialized parallel computing platforms. For example, graphic processing units (GPUs), IBM's Cell processors, and field-programmable gate arrays (FPGAs) are all popular platforms suitable for parallel data processing. These parallel platforms can be roughly divided into four groups based on the level of parallelism supported, bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism. The third approach is hybrid computation, a combination of the previous two approaches. The computation tasks can be roughly divided into two types: serial tasks and parallel tasks. Intrinsically serial tasks are suitable for single-core CPUs running at a high frequency, while parallel tasks can benefit greatly by allocating them to multiple accelerator cores. Most applications are usually a mixture of such serial and parallel tasks, and the best way to boost performance is by utilizing heterogeneous or hybrid architectures [19]. A hybrid system, with a heterogeneous combination of processing elements, has the advantages of both general purpose CPUs and other processing platforms. An algorithm can then be partitioned into different sections based on its computational profile. The computation-intensive parallel operations can be mapped onto the

parallel processing platforms, while leaving complex control logic and sequential computations to the general purpose processors.

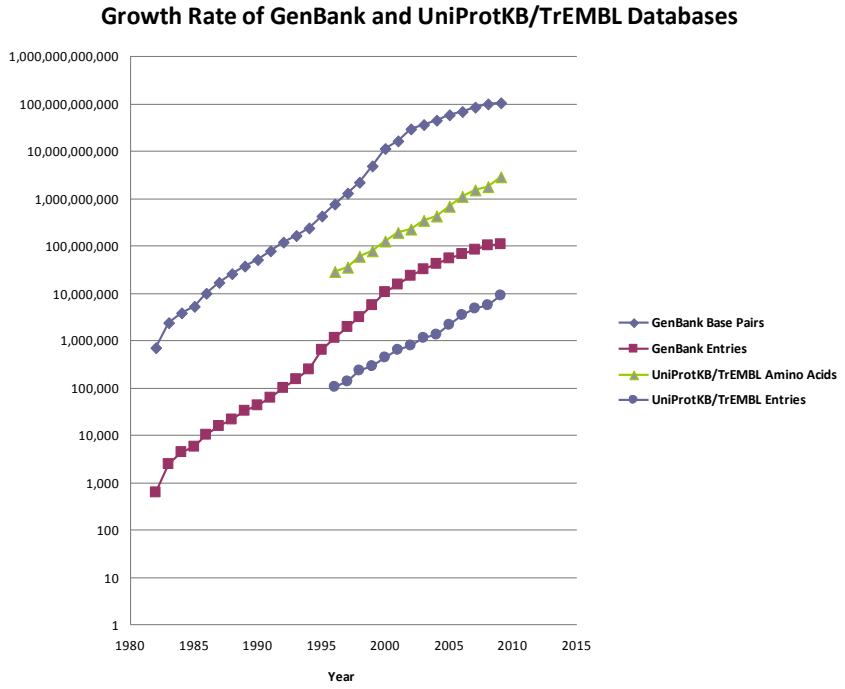


Figure 1.1 Growth of databases [91]

The rapid growth of the biological databases, and the urgent need to accelerate the computational analysis involving these databases, using commodity computing resources (as highlighted above) provides the motivation for the work presented in this thesis. To provide a manageable focus, we narrow down our study to consider two fields in bioinformatics: genome search and short read alignment.

A genome search identifies global or local similarities between two DNA sequences: a query sequence and a subject sequence. The general approach for the sequence alignment computation is based on dynamic programming algorithms, which solve a complex computation problem by finding optimal answers for its sub-problems with smaller scale. For two sequences of length m and n, the search space for the conventional dynamic programming algorithm is $m \times n$. If a genomic DNA sequence consists of several billion bases, the search space will be extremely large and impractical for modern processors. Under such circumstances, many

heuristics have been proposed to accelerate the computation. Different heuristics provide different runtime performances, but with differing alignment accuracy. Among them all, FASTA [102] and BLAST [8] are two important heuristics widely used by the biology research community to solve sequence alignment problems. In this thesis, we propose an FPGA-favorable architecture to accelerate the most time-consuming stage of the NCBI BLASTN algorithm. The proposed architecture achieves high computation efficiency while maintaining a similar sensitivity to the original BLAST algorithm.

The second field that we consider, short read alignment, can be treated as a special case of the genome search. The query sequence is changed from a long consecutive sequence into thousands or millions of short sequences (short reads). Short read alignment can also be computed utilizing the BLAST algorithm, but it is not a highly efficient heuristic to solve the short read mapping problem. Many tools have been proposed for this particular problem, such as Bowtie [65], BWA [67], SOAP [73, 74, 78], and so on. However, along with the rapid growth of the short read datasets, the requirement for a short read aligner with faster processing speed and high sensitivity will always exist within the bioinformatics community. As the computations among different reads are independent of each other, massive parallel computation can thus provide higher computation efficiency in solving such a problem. Again, to solve this problem, we choose FPGAs as our experiment platform due to their fine-grained pipelining and parallelism. A computationally efficient architecture is designed to accelerate the computation-intensive part of the short read alignment problem and also to maintain a high sensitivity and accuracy.

The algorithms applied in both the genome search and the short read mapping involve intensive computations (sometimes also intensive memory accesses). The easiest way to improve the performance of these algorithms on a general purpose processor is to increase the clock frequency of the CPU. Alternatively, another solution is to increase the number of processor cores. However, while multi-core processors are able to provide an immediate performance improvement, there have been many examples showing that an algorithm, appropriately optimized for an

architecture with a higher degree of parallelism, can provide better performance [13, 27, 55, 123]. Heterogeneous computing systems provide yet another alternative solution for computations with little or no data dependency. Examples of such architectures include: Cell Broadband Engines (Cell/BEs), compute unified device architecture (CUDA)-enabled GPUs, or FPGAs, all of which can achieve a higher degree of parallelism than the current generation general purpose processors.

1.1.1 Cell/BE

The Cell/BE [28, 29] is a multi-core microprocessor architecture which is designed to provide more powerful computation capabilities than the conventional desktop processor. Normally, a Cell processor consists of four components: the external input and output structure, the main processor, named the Power Processing Element (PPE), eight co-processors, called the Synergistic Processing Elements (SPE) suitable for compute-intensive workloads, and a specialized high-bandwidth circular data bus called the Element Interconnect Bus (EIB). The Cell/BE is optimized to conduct single precision floating point computation. The SPEs are reduced instruction set computing (RISC) processors supporting 128bit single instruction multiple data (SIMD) operations for single and double precision instructions [45]. For a Cell processor running at 3.2GHz, the EIB has a theoretical peak bandwidth greater than 300GB/s [103]. The Cell/BE provides faster computation capability than state-of-art CPUs. Each core within the Cell/BE can run a separate program, which provides a higher degree of flexibility. Its memory coherence architecture, high-bandwidth data bus, and powerful peak computational throughput make the Cell/BE suitable for accelerating applications in the fields of multimedia, vector processing, supercomputing, cluster computing, distributed computing, bioinformatics, and so on. Compared to a CPU or GPU, it is slightly more difficult to program and is relatively more expensive, both of which limit its widespread usage.

1.1.2 GPU

CUDA, initially developed by NVIDIA for graphics processing, is a popular architecture for high performance computing. Meanwhile, it is also a scalable parallel programming model and a software environment for various parallel computing applications [94]. Developers only need to focus on the parallel algorithm design with a minimal extension to the conventional C/C++ programming language, rather than using a new parallel computing language. One key feature for CUDA-enabled GPUs is that the computation task is executed by many concurrent threads, which enables massively parallel computing for data-independent applications. Furthermore, CUDA provides a fast shared memory that can be shared among threads, which can further improve the performance of CUDA-enabled GPUs. For a Tesla C2050 GPU [122], it provides 448 CUDA cores with a maximum 1.03TFLOPS performance for single precision floating point computation, 3GB GDDR5 memory, 144GB/s bandwidth for memory access, and a Peripheral Component Interconnect Express (PCIE) x16 Gen2 bus to connect to the CPU. Independent data can be stored to the GPU's memory through direct memory access (DMA). Computation is conducted by the core arrays supporting massive parallelism. GPU memory provides a faster access speed than the main memory, which can further improve the computation speed. All these merits make a CUDA GPU a suitable platform for applications in the field of Computational Biology (CB). Generally, GPUs perform very well when it comes to single precision floating point arithmetic. The GPUs are quite suitable for processing a large set of independent data with the same operation. However, when extensive synchronization and communication are required by the algorithm, the performance improvements are diminished [19].

1.1.3 FPGA

FPGAs are integrated circuits designed to support customized configuration after manufacturing. The designer generates an FPGA bit-stream, usually derived from a hardware description language (HDL) representation of the required design, to

program FPGA functionally. The two most popular HDLs are VHDL [126] and Verilog [125]. The design flow for an FPGA application is somewhat different to that of a conventional software application, and is shown in Figure 1.2. Take Xilinx design flow as an example. The designer first creates the system design files using a HDL and assigns a user constraint file (UCF) to further specify the design. This constraint can be pin allocation, timing constraint, or area constraint. After that, the design files are converted into a device level netlist using a synthesis tool (shown as the second stage in Figure 1.2). Before implementation, a simulator is used to check the correctness of the design's functionality. Usually, this simulation stage is called functional simulation. The implementation stage can be further divided into three steps: Translate, Map, and Place & Route (PAR). The Translate step merges all of the input netlists and design constraints and generates a Xilinx native generic database (NGD) file. The logical design is thus represented using Xilinx primitives in the NGD file. The Map step converts the NGD file to FPGA physical components, such as slices, IO blocks (IOBs) and so on. The Map step outputs a native circuit description (NCD) file with precise switch delay information. The PAR step assigns a location in the FPGA for each component and connects them with interconnects. The propagation delay information is also added to the NCD file during PAR processing. Before undertaking board level testing, post-route simulation is used to examine the functional correctness and timing of the design. In fact, the simulation process is an iterative process. If any of the simulation processes identify errors in the implementation, the designer need revert to the first design stage to correct the identified errors. Once the simulation errors and user constraints have been satisfied, the next step is to generate a bit-stream file to configure the FPGA device. The bit-stream file can be downloaded directly to the FPGA, or converted to a PROM or JTAG file, to configure the device. Device testing is then performed. Unlike GPUs which provide parallelism among the processor arrays, FPGAs support massive fine-grained parallelism and pipelining at the circuit level. Modern FPGA architectures further boost their performance by inserting fundamental blocks, such as high speed memories, high performance multiplier cores, and so on. A more detailed introduction on FPGA's

features is presented in Chapter 2. Generally speaking, FPGAs are suitable for a wide range of high throughput applications, such as digital signal processing, aerospace and defense systems, bioinformatics, medical imaging, telecommunications, and so on.

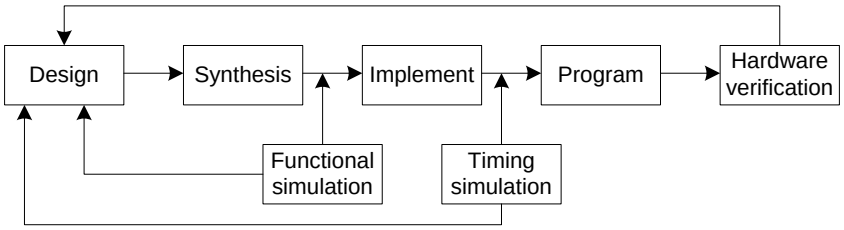


Figure 1.2 FPGA design flow

Although current FPGA chips can support floating point operations, it is difficult to quantify an FPGA's performance in terms of floating point operations, as the architecture is not optimized for floating point operations. The most powerful computation capability that FPGAs can provide lies in fixed point, integer, or bit-operations [19]. The advantage of FPGA is in its high data throughput and relatively low power consumption, but the FPGA's price and development effort (particularly the effort needed to learn hardware design using a HDL and the use of the design tools) means they are used in niche areas, rather than as mainstream computing engines. An architectural comparison between CPU, GPU, Cell/BE, and FPGA, reproduced here as Table 1.1, is presented in [19]. The selected devices are all released in 2008.

Table 1.1 Summary of architectural properties[1] [19]

	CPU	GPU (AMD/NVIDA)	Cell/BE	FPGA
Full cores	4	-	1	2
Accelerator cores	0	10/30	8	2016 DSP slices
Intercore communication	Cache	None	Mailboxes	Explicit wiring
SIMD width	4	64/32	4	Configurable
Float operations per cycle	16	1600/720	36	520
Frequency (GHz)	3.2	0.75/1.3	3.2	<0.55
Single precision performance	102.4	1200/936	230.4	550
Power Consumption (Watt)	130	220/187.8	90	4.3[2]
Gflops/watt	0.8	5.5/5	2.5	13.7
Mflops/USD	70	800/550	>46	138
Accelerator bandwidth (GB/s)	N/A	109/102	204.8	N/A
Main memory bandwidth (GB/s)	25.6	8	25.6	6.4
Maximum memory size (GB)	24	2/4	16	System dependent

1. The numbers are of current hardware, and are reported per physical chip. The CPU is an Intel Core i7-965 Quad Extreme, the AMD GPU is the FireStream 9270, the NVIDIA GPU is the Tesla C1060, the Cell/BE is the IBM QS22 blade, and the FPGA is the Virtex-6 SX475T. 2. The number is the device static on-chip power got from Xilinx Power Estimator (XPE).

1.2 Objectives

Despite some of the disadvantages, FPGAs have become an increasingly attractive platform for high performance computing (HPC). A number of compute-intensive applications have been accelerated using FPGAs, including some from the bioinformatics domain [60]. This has provided us with the motivation to further

investigate the FPGA acceleration of algorithms widely used by the bioinformatics community, since many of them are computationally intensive with a parallel data structure.

Unlike general-purpose processors executing each instruction in a sequential manner, FPGAs can provide a finer degree of parallel computation due to their architecture (computation is conducted through circuits rather than instructions). However, the algorithms or heuristics widely used in bioinformatics are initially designed for software implementation. Thus, we need to modify the original algorithm to an FPGA-favorable version with a similar functionality. In certain cases, the FPGA architecture alone is not appropriate to efficiently implement the complete functionality. Therefore, a general-purpose processor is also required in the final design resulting in a hybrid system. The design procedure for our hybrid system firstly requires that we apply runtime profiling to locate the computationally intensive parts of a bioinformatics algorithm. Based on the profiling results, the algorithm is then partitioned into the software and hardware parts. A computationally efficient architecture is proposed for the hardware part and the control is left to the software part. The final design is then compared with the original algorithm. As bioinformatics covers a very large scope of research topics, it is impossible to propose an architecture suitable for all applications. Instead, we present an in-depth analysis of two specific problems to see if it is possible to improve their performance. Our research is thus focused on the efficient FPGA implementations of the following sequence alignment problems in bioinformatics:

- *Genome search*: The genome search is one of the fundamental tasks in bioinformatics research as similarities between sequences can indicate similar function or structure. Many different methods for DNA sequence analysis exist today. Among all these search methods, BLAST is probably one of the most widely used tools for genome search. It consists of different filtration algorithms to mitigate the burden introduced by the intensive computations. However, as the volume of genetic database increases dramatically, BLAST becomes less efficient. Thus, there is an urgent need

for a faster solution to this genome search problem. As FPGAs have shown outstanding capability for parallel data processing, we believe that a highly efficient parallel processing structure on FPGAs can effectively improve the performance of a genome search.

- Short *read alignment*: Next generation sequencing (NGS) is a new technology to overcome the barriers of deciphering DNA sequences. The appearance of NGS technologies has triggered ground-breaking discoveries and has stimulated a revolution in genomic research. Based on this sequencing technology, many hundreds of thousands or millions of short reads (of size from 25-500bp) can be generated in a relatively short time. Subsequently, this data must be reconstructed to infer the original DNA target, either through assembly or alignment to a reference genome. As one important branch derived from NGS technology, the short read alignment problem has generated considerable interests. Short read alignment requires data reconstruction of a high volume data set, which is a time-consuming process. In addition, it is computationally intensive and the memory overhead can also be very large (depending on the reference genome and the short read dataset). For general purpose processor based solutions, a tradeoff is made to balance the computation complexity and the alignment sensitivity. As the short reads are independent of each other, FPGAs are suitable platforms for conducting the alignment operations in parallel. Improving the computation speed is only one of the constraints for the short read alignment problem. To maintain a similar, or a better, sensitivity compared to the original algorithm is the other constraint, both of which are important in the field of DNA sequencing.

1.3 Contribution

As mentioned above, we investigate two problems in sequence alignment. We have designed and implemented specific architectures for each of them. Performance comparisons are also provided to give a comprehensive analysis between our

design and other popular tools in this field. The primary contributions of this thesis are summarized as follows.

1. A reconfigurable accelerator for the word-matching stage of BLASTN: as the word-matching stage computation is the most time-consuming part of the NCBI BLASTN algorithm, acceleration of this stage can largely shorten the overall execution time. In this thesis, we present a computationally efficient architecture for the word matching computation on an FPGA. The proposed architecture achieves a faster processing speed while maintaining a similar sensitivity in comparison to the original BLASTN algorithm. Experiments under different test conditions indicate the performance bottlenecks for the current design. The influence of different architecture configurations is also examined in this thesis. The performance analysis on different design factors (the number of hash functions, the degree of parallelism, and the false positive probability) can be used as a guide for future architecture design.
2. A hybrid short read aligner:

   In this thesis, we design and implement a short read aligner on an FPGA based on the "seed and extend" strategy. The computation is partitioned into the software part and the hardware part according to the runtime profiling conducted in advance. A parallel block-wise alignment core is proposed to approximate the conventional banded Needleman-Wunsch algorithm. Different partitioning methods have been examined, which leads to performance variations in the final system. Performance evaluation, using simulated as well as real datasets, shows that the proposed architecture achieves a faster processing speed and a better alignment quality for both single-end and pair-end alignments compared to two CPU-based aligners: GASSST and BWA. The analysis also indicates the performance bottlenecks for the current design.

1.4 Organization

The rest of the thesis is organized as follows.

1. Chapter 2 gives a brief introduction to both reconfigurable computing and biological sequence analysis.
2. Chapter 3 provides an introduction to genome search and reviews the past implementations on this research topic.
3. Chapter 4 presents the word matching accelerator on FPGA with a detailed investigation of the architecture design and performance analysis.
4. Chapter 5 reviews the basics of the short read alignment problem and the short read aligners on different platforms.
5. Chapter 6 introduces the hybrid short read aligner design and analyzes the performance difference under different partitioning strategies.
6. Chapter 7 concludes the thesis and suggests possible avenues for future investigation.

Chapter 2

FPGA and Its Applications in Bioinformatics

2.1 FPGA technology

For conventional computation methods, almost all can be divided into two categories, computation based on hardware circuits and computation based on software programs. A popular example of the first category is application specific integrated circuits (ASICs). ASICs are integrated circuits customized for a particular application rather than intended for general-purpose usage. They can achieve very fast processing speed with high computation efficiency and low power consumption. Despite their obvious advantages, ASICs also have several drawbacks that hinder their utilization in this area. The other category, software program-based computation, is far more flexible. A computation operation is accomplished by executing a set of instructions. By altering the composition of the instructions, the functionality of the system is changed without any modification to the physical structure of the processor. The price paid for this flexibility is far lower performance and less efficiency than ASICs. Instead of conducting the computation directly on the circuits, the computation converts to three steps: instruction fetching, instruction decoding, and execution. The resources on a general-purpose processor are reused in time, following these three steps repetitively. This results in extra computation overhead.

Another platform which provides some of the benefits of both hardware circuits and software programmability is the FPGA. Generally speaking, FPGAs can provide a better performance than the software program, while also achieving a higher degree of flexibility than ASICs, which makes it one of the most widely

used reconfigurable devices nowadays. A typical FPGA architecture consists of different types of programmable functional blocks. These include configurable logic blocks (CLBs), embedded multipliers, on-chip memory blocks and programmable I/O blocks. The CLBs can be used to implement various logic operations. The embedded multipliers optimize the multiply operation to improve design performance. The on-chip memory blocks can provide fast memory access. The I/O blocks connect the chip to the peripheral components. All these functional blocks are connected through the programmable routing fabric. In practice, the design tools (e.g. ISE for Xilinx or Quartus for Altera) analyze the design described using HDLs, and decompose it into different basic functional blocks targeted to a specific FPGA device. The design tool will also determine the optimal interconnect path for all these blocks. The same FPGA device can be programmed repetitively, which provides a great flexibility on design modification on FPGAs.

FPGA technology has a number of advantages in terms of performance, time to market, and cost [89].

- **Performance**: as the only limitation on the number of "processing cores" on an FPGA is the physical size, designers can improve a design's performance by simply duplicating the circuits with the same functionality and then execute them in parallel. Fine-grained pipelining and massive parallelism take full utilization of the "inherent parallel" attribute of circuits. Meanwhile, current high speed I/O blocks attached to the FPGA chip can further improve a design's throughput.
- **Time to market**: the flexibility that FPGAs provide shortens the prototype time. New designs and architectures can be implemented and tested on the same FPGA chip without going through the long fabrication stage of an ASIC design. Furthermore, a large number of commercial IP cores can save a great portion of the development time and provide guaranteed performance at the same time.
- **Cost**: the non-recurring engineering (NRE) cost for FPGA design is much smaller than that of an ASIC design. The requirement of a digital system

might change over time and the customer would like to make incremental changes to the functionality. For ASIC solutions, this means a completely new design, as it is impossible to change the inner structure of an ASIC; however, for FPGA solutions, it is possible to make minor changes and download new programming data (a bitstream) to the FPGA chip. While an individual FPGA is relatively costly, for small to medium volumes, this cost is negligible compared to that of ASIC.

Xilinx and Altera are the two major vendors in the FPGA industry. Together, their products have over 80% market share. Like other computing devices, FPGA products have evolved over time. For example, the Xilinx Virtex FPGA family (from Virtex-2 to Virtex-7) is shown in Table 2.1.

Table 2.1 Xilinx Virtex-series specifications*

	Virtex-II	**Virtex-II pro**	**Virtex-4**	**Virtex-5**	**Virtex-6**	**Virtex-7**
Slices	46,592	44,096	89,088	51,840	118,560	178,000
BRAM (Kbits)	3,024	7,992	6,048	10,368	25,920	67,680
Max I/O	1,108	1,164	960	1,200	1,200	1,100
DSP slices	N.A.	N.A.	96	192	864	3,360
Freq (MHz)	650	1050	1028	1098	1098	1818

* The Virtex-II series specifications refer to the XC2V8000, XC2VP100, XC4VLX200, XC5VLX330, XC6VLX760, and XC7VX1140T, respectively. The devices chosen here are the largest designs in their family. The frequency value listed is the toggle frequency for CLB switching of each device at the lowest speed grade.

Table 2.1 shows that, from Virtex-II to Virtex-7, the volume of slices and on-chip memory has increased dramatically. Note that the Virtex-5 FPGA reports fewer slices than the Virtex-4 device. This is because a different slice organization has applied since Virtex-5 (i.e. each slice in a Virtex-5 FPGA contains 4 6-input lookup tables (LUTs) and 4 flip-flops; in contrast, the earlier generation slice contains 2 4-input LUTs and 2 flip-flops). A LUT is one of the basic components in an FPGA, used to implement combinational logic as truth tables. As more resources are available on a single FPGA chip, we can design and implement more complicated applications on a single FPGA device, which leads to greater

improvements in the FPGA's computation capability. The internal frequency has also improved after decades of FPGA development, but at a much slower speed comparing to that of the available resources. Besides expanding the volume of resources on a single chip, the FPGA vendors also integrate popular functional units as hard macros to better meet different application requirements. Many signal processing algorithms involve intensive multiply-accumulation computations. Prebuilt multiplier-accumulate circuits (known as DSP slices) are integrated to improve the performance and reduce logic consumption for this specific operation. Block RAMs (BRAMs) are another important on-chip resource, which can be the key factor for FPGA selection. The BRAM resources are integrated in blocks of 18kbit or 36kbit on-chip, which provides fast memory access and can be fully customized by the user. There are also other hard macros within the FPGA chip to provide design convenience under different circumstances, such as digital clock management (DCM) for clock control, PCIe blocks for high speed data transfer, and so on.

The development of FPGA products not only improves their performance but also boosts the appearance of new design concepts, such as system-on-a-programmable-chip (SoPC). The processor core (as either a "hard" or a "soft" CPU) within a SoPC system provides extra flexibility to the FPGA device with only a slight performance sacrifice. The hard processor core uses dedicated silicon, while the soft core uses programmable FPGA resources to implement its functionality. The soft core can be fully tailored by the designer (e.g. customize the memory address, number of peripherals, and so on) to meet different embedded system requirements. Two examples of a soft core processor are Altera's Nios II and Xilinx's MicroBlaze. As the soft core processors are implemented using FPGA resources, their highest working clock frequency is limited by the FPGA technology.

Another design concept for FPGA is to treat it as a co-processor (similar to GPU accelerator cards) attached to a general-purpose computer system leading to the heterogeneous computing model. The combined CPU-FPGA computing platform presents a number of opportunities. The general-purpose multi-core CPU,

operating at a high clock frequency (over 3GHz) is suitable for sequential multi-threaded applications; while FPGAs, on the other hand, have a fine-grained parallel architecture, which can provide more efficiency for highly parallel operations. Thus, the heterogeneous model takes advantages of both general-purpose CPUs and FPGAs. A common computation task needs to be split into the software components and the highly parallel hardware components. These are then mapped to the different computing platforms. The communication between the two platforms can be accomplished using modern high speed data transfer interfaces, such as PCIe, HyperTransport, Ethernet, and so on. The heterogeneous model can provide a high performance solution for a target application, but more design effort is required during the development process. The final system's performance depends on finding an appropriate partition to the original computing task.

In general, FPGAs operates at a relatively low frequency (compared to CPU or ASICs), but their architecture is quite suitable for massive parallel data processing. As many bioinformatics problems involve computation-intensive operations, FPGAs provide the opportunity to optimize these operations. Some of the attractive aspects that FPGAs provide in this area include:

- A fine-grained pipelined architecture is suitable for mapping the searching algorithms used in computational biology.
- A large number of IP cores supporting full customization can largely reduce the design time and guarantee the performance of bioinformatics algorithms.
- High-level design methodologies use hardware description languages (HDLs) to speed up the design process and enable reuse of key modules across different bioinformatics applications sharing similar functionalities.
- Newer devices based on the latest manufacturing technologies with better performance characteristics are regularly being introduced to the market. This provides an immediate performance boost as HDL-based design can be mapped to these latest devices with minimal effort.

- Devices are becoming cheaper every year. This allows platforms based on these devices to be used by an ever-increasing number of bioinformatics labs.
- The raw data used for a DNA sequence representation can be coded using a 2bit format. The fundamental operations typically based on character comparison can be more efficiently implemented on FPGA by customizing the data path width.
- A DNA sequence can be transferred to the FPGA components in streaming format which is suitable for the spatial and temporal parallelism within the FPGAs.

Bioinformatics is an interdisciplinary research field that uses computational methodologies or algorithms in computer science to analyze information gained from biological origins or to solve particular biology problems. Since it is closely related to computer science, tools developed for different bioinformatics applications are usually built on general-purpose processors. Along with the rapid growth of biological datasets, these software-based tools are facing greater pressure to process this large volume of data. FPGAs can be a competitive platform to solve some of the biological problems. In fact, we have already seen many FPGA applications in various research areas in Bioinformatics. Some of these are briefly described in the following sections.

2.2 Sequence analysis using FPGA

In Biology, DNA sequences are analyzed to find out motifs, repetitive sequences, regulatory sequences, and RNA genes. As the original data is presented using characters, sequence analysis can be viewed as the fundamental method to understand the relationship between different sequences, which also has a significant influence on other research areas in Bioinformatics. For example, an FPGA-based framework for pair-wise sequence alignment [13] has been designed, which can be parameterized to support different alignment algorithms, such as the Smith-Waterman algorithm [113] and the Needleman-Wunsch algorithm [93].

Over 100 processing elements (PEs) are implemented to improve the performance of pair-wise comparison. This design reports a 62× speedup compared to the equivalent software implementation running on a general purpose PC. The Convey GraphConstructor (CGC) [27] is a commercial product designed for sequence assembly using both FPGAs and general-purpose CPUs. It optimizes the memory architecture for the intensive random memory access and shortens the computation time with the help of multiple FPGAs. CGC achieves a 6.2× speedup and reduces RAM consumption by 76%. An FPGA-based coprocessor to solve the gene identification problem was designed in [26] based on the Glimmer algorithm [33]. It presents a tree structure to implement the computation intensive part of the algorithm on the FPGA and achieves a 2.37× speedup compared to the CPU program. The process of finding protein motifs is accelerated in FPGA in [55] by utilizing Hidden Markov Models (HMMs) [61]. The HMMer engine implemented on the FPGA utilizes a systolic array structure to enhance the comparison operations achieving a 190× speedup over the same computation running in a general purpose CPU. We are also interested in improving the performance of sequence analysis algorithms using FPGAs and a more comprehensive literature review is given in Chapter 3 and Chapter 5, respectively.

2.3 Evolutionary biology using FPGAs

Evolutionary biology studies the evolutionary relationship among different species. Research in evolutionary biology includes phenomena explanation, phylogenetics, genetic architecture, and evolutionary synthesis. The study of evolutionary biology helps us understand the origin of species and predict possible changes based on existing population models. A co-processor structure is developed by [12] to accelerate median-based phylogenetic reconstruction, where the computationally expensive part (the break median core) is accelerated using an FPGA. This design achieves a speedup ranging from 5× to 189× under different test conditions. A different parallel architecture proposed by [5] accelerates the computation of the phylogenetic maximum likelihood function [39]. On-chip DSP resources are utilized to implement the "Basic Cells" of the FPGA core. Compared to the

software program running on a single core, this design achieves a speedup of 13.68. Another heterogeneous design [137] that aims to accelerate the phylogenetic likelihood function is built within the MrBayes [109] framework. This hybrid design implements multiple float-point operations in parallel and provides a 10× speedup relative to the software program. Alachiotis *et al*. [6] proposed a hybrid system to accelerate the computation of the discrete parsimony function. 512 processing units (PRUs) are implemented on a Xilinx Virtex 6 FPGA achieving a speedup factor of 9.65.

2.4 Gene expression analysis using FPGAs

Gene expression is a process whereby a gene is synthesized to a functional gene product. Depending on the different types of gene, the functional gene product can be protein or functional RNA. The gene expression process includes several steps, such as transcription, RNA splicing, translation, and post-translational modification. By studying the gene expression, we can understand the function of a particular gene which can then be used to explain the cause of diseases, such as cancer. The inherent parallelism of the computation-intensive Bayesian learning method [42] (used for the reconstruction of gene regulatory networks) is explored by [105]. The reconfiguration capability is utilized to accomplish the network node score computation and the network structure update iteratively on the same FPGA. The performance evaluation demonstrates a speedup of 76 times over the software implementation. A parallel architecture on an FPGA platform [52] is introduced to accelerate the K-means clustering algorithm [82] which is widely used for Microarray data analysis. The 5-core FPGA design reports 51.7× speedup and 206.8× energy efficiency.

2.5 Protein structure prediction using FPGAs

As an important branch in Bioinformatics, protein structure prediction is used to predict the structure of an amino acid sequence which can then be used to understand the function of the corresponding protein. Protein structure prediction has wide application potential, such as in drug design in the medicine industry or

novel enzyme design in biotechnology. For example, a fine-grained FPGA accelerator [133] is proposed to compute the Garnier-Osguthorpe-Robson (GOR) algorithm [43] for the secondary protein structure prediction. The evaluation experiments demonstrate a speedup factor of 430× over the original algorithm and 110× over the multi-thread software implementation. Meanwhile, its power consumption is only 30% of that of the CPUs. Another CPU-FPGA co-design [56] maps the most computation intensive operations (float point arithmetic operations) onto an FPGA to reduce the overall computation time for the protein energy minimization algorithm. For a single FPGA board, it achieves 5 times speedup compared to the software program.

2.6 Summary

Since the FPGA's inception in the early 1980s, its performance has increased dramatically, with this trend likely to continue into the future. The FPGA's outstanding computation capability provides opportunities to solve a range of different kinds of bioinformatics problems. In this chapter, we have presented a brief introduction to some of the existing FPGA-based solutions in the different bioinformatics fields. This brief overview shows that there are many possible avenues where FPGA based parallel computation could be applied. As sequence analysis is fundamental to bioinformatics research, it has received more attention. In fact, sequence analysis also includes many sub-problems which could be investigated. To limit our research scope, we choose two important sequence analysis problems as case studies to explore the capability of FPGAs to boost the performance of bioinformatics applications: genome search and short read alignment, both of which consists of computation-intensive operations making them good candidates for FPGA acceleration. The details are presented in subsequent chapters.

Chapter 3

Genome Search and its Implementation

3.1 Background

In modern molecular biology and genetics, a genome is a sequence, encoded either in DNA or RNA, containing an organism's entire hereditary information. For a DNA sequence, it consists of alphabets represented by four letters $\sum$ = {A, C, G, T}, where each character represents **A**denine, **C**ytosine, **G**uanine, and **T**hymine, respectively. RNA is the transcription of a DNA sequence. After the translation process, the RNA can convert to proteins. For a standard protein sequence, it consists of 20 amino acids with letters $\sum$ = {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}. However, a common question exists among biologists: how to effectively evaluate sequences with a large volume of characters to explore their function, structure, or evolutionary attributes from a biological point of view? Under such circumstances, sequence analysis methods [35] became a popular study area in Bioinformatics. The study of biological sequence analysis includes a very wide range of relevant topics, such as pair-wise sequence alignment, multiple sequence alignment, motif finding in DNA sequences, and so on. The difference lies in that the former algorithm only compares two sequences at a time, while the latter one compares multiple sequences in a single run. A common usage for pair-wise sequence alignment is to identify homologous sequences (i.e. to locate all regions in a genome database similar to the query sequence). Sequences that show a high degree of similarity usually have a similar genetic function, which is very important for biologists to establish links between the new gene and known genes. Figure 3.1 shows an example for the pair-wise alignment. The "-" indicates a gap between two sequences.

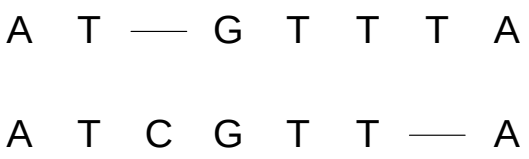


Figure 3.1 Pair-wise alignment for two sequences of length 7

Multiple sequence alignment has a slightly different application background. For example, sequences from different species may not exhibit a high degree of similarity biologically. Pair-wise alignment might fail to identify biologically important sequences with weak pair-wise similarities, but simultaneous comparison among many sequences often allows one to find similarities. Figure 3.2 shows an example of multiple alignment of three sequences, where "|" indicates an exact match and the unaligned character pairs indicate the appearance of mismatch. Although multiple sequence alignment is an extension of the pair-wise alignment, its computation is far more difficult, sometimes leading to NP-complete optimization problems [129].

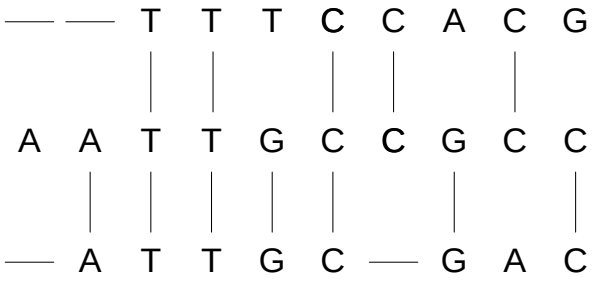


Figure 3.2 Multiple alignment of three sequences

A sequence motif is a pattern that is widespread and has a biological significance among multiple nucleotide or amino-acid sequences. Motif finding can be treated as an algorithm to find short substrings that occur surprisingly often among a given set of sequences from a genome, which is important to understand the mechanisms that regulate gene expressions by identifying Transcription factor binding sites (TFBSs). Transcription factors bind themselves to regulatory regions (located

upstream of the gene). Relying on a single string to represent a motif often fails to represent the natural mutations appearing in real biological sequences. A more flexible representation of a motif is to use the profile matrix. A typical case of motif finding is shown in Figure 3.3. The substrings of length k (here, $k = 8$) from different positions of the original DNA sequence form the alignment matrix. The profile matrix illustrates the variation of nucleotide composition at each location of the substring. The consensus string represents the most popular element in each column of the alignment matrix.

Alignment		A	T	C	C	A	G	C	T
		G	G	G	C	A	A	C	T
		A	T	G	G	A	T	C	T
		T	T	G	G	A	A	C	T
		A	A	G	C	A	A	C	C
		A	T	G	C	C	A	T	T
		A	T	G	G	C	A	C	T
Profile	**A**	5	1	0	0	5	5	0	0
	T	1	5	0	0	0	1	1	6
	G	1	1	6	3	0	1	0	0
	C	0	0	1	4	2	0	6	1
Consensus		A	T	G	C	A	A	C	T

Figure 3.3 The alignment matrix, profile matrix and consensus string

There are still more research topics in the field of genome sequence analysis. In this thesis, we focus our research on solving the pair-wise sequence alignment problem. In the subsequent sections, a more detailed introduction to pair-wise sequence alignment algorithms and related implementations will be presented.

3.2 Pair-wise alignment basics

Several well-known aligning methods exist for the pair-wise sequence alignment. The dot-matrix approach [85] is a qualitative, simple and straight forward method. A dot is inserted into the matrix constructed by the sequences to represent a

possible match. The dot plot for sequences with high degree of similarity will show a single line along the matrix's main diagonal. However, the dot plot only provides a directly impression on the similarity between two sequences, it fails to quantify the degree of similarity. The easiest way to get a quantitative result is to compute the Hamming distance [46] (i.e. identify the number of different characters). The Hamming distance computation is simple, but it is not enough to disclose the true relationship between two sequences. It has several drawbacks for comparing DNA sequences: the Hamming distance calculation rigidly assumes that two sequences are of the same length and the i^{th} character of one sequence is already aligned to the i^{th} location of the other. However, mutation in the DNA sequence is an evolutionary process. DNA replication errors cause three types of error: substitution, insertion, and deletion of nucleotides. As insertions and deletions (indels) exist, there is no guarantee that the i^{th} character in one sequence corresponds to character of the same location in the other sequence. Under such circumstances, edit distance is applied to evaluate the similarity between two DNA sequences, which also allows sequences of different length. Although the details of the algorithms are slightly different across various applications, the basis for edit distance computation is the dynamic programming (DP) algorithm. For example, Figure 3.4 (a) shows the direct comparison for two 7bp sequences with a Hamming distance of 4; in contrast, when indels are allowed, the edit distance for the same sequence pair is 3, as shown in Figure 3.4 (b). Comparing the two Figures, it is clear that the search space for the edit distance is much bigger than that of the Hamming distance. In Figure 3.4 (b), its search space is the product of the two sequences.

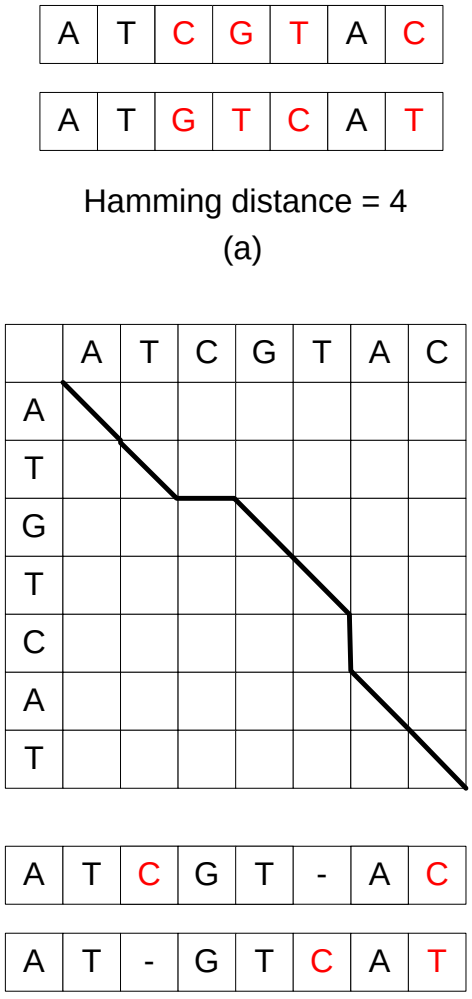


Figure 3.4 (a) Hamming distance (b) edit distance

In practice, merely computing the number of edit operations is not enough to evaluate the degree of similarity between two DNA sequences. A scoring mechanism is applied in the alignment process to get the optimal alignment result. The scoring mechanism is quite different for DNA sequence alignment and protein sequence alignment. For the former, a series of scores are utilized (e.g. a penalty score for a substitution error; a bonus score for a match; a penalty score for a gap opening; a penalty score for a gap extension). For the latter, a more complicated substitution matrix is applied for the substitution errors. Two most widely used substitution matrices are the point accepted mutation (PAM) matrix [30] and the

block substitution matrix (BLOSUM) [48]. The PAM matrix and BLOSUM reflect the frequency that an amino acid is replaced by another amino acid during evolution. As, in this thesis, we are mainly focused on the computation of DNA sequences, the detailed introduction on the PAM and BLOSUM is omitted here. Normally, the two most widely used dynamic programming algorithms are the Needleman-Wunsch (NW) algorithm [93] for the global sequence alignment, and the Smith-Waterman (SW) algorithm [114] for the local sequence alignment.

The general form of the global alignment problem is described below.

Global Alignment Problem:

Align two sequences to get the best global score constrained by a given scoring matrix.

Input: Two strings (*v* and *w*), and a scoring matrix δ.

Output: An alignment which maximizes the alignment score among all possible alignments of *v* and *w*.

The corresponding alignment score $S_{i,j}$ of an optimal alignment between the i^{th} location of *v* and the j^{th} location of *w* is as follows:

$$S_{i,j} = max \begin{cases} S_{i-1,j} + \delta(v_i, -) \\ S_{i,j-1} + \delta(-, w_i) \\ S_{i-1,j-1} + \delta(v_i, w_j) \end{cases} \quad (3.1)$$

The scoring matrix can be different according to specific applications. However, in certain cases in molecular biology, the similarity between two sequences only exists in a short region and is quite different over the remaining regions. Under such a situation, the global alignment cannot find the homologous region as it tries to align the entire sequence. When biologically important similarities only exist in a certain region of DNA segments, biologists use local sequence alignment to conduct the computation. The general form of the local alignment problem is given below.

Local Alignment Problem:

Find the best substring alignment between two sequences.

Input: Two strings (*v* and *w*), and a scoring matrix δ

Output: An alignment which maximizes the global alignment score among all the substrings of *v* and *w* using the score matrix δ

Although the local alignment problem is seemingly more complicated than the global alignment problem, there is only a small difference in their mathematic representation (see Equation 3.2.) and they also share the same search complexity.

$$S_{i,j} = max \begin{cases} S_{i-1,j} + \delta(v_i, -) \\ S_{i,j-1} + \delta(-, w_i) \\ S_{i-1,j-1} + \delta(v_i, w_j) \\ 0 \end{cases} \quad (3.2)$$

Generally, the dynamic programming based alignment algorithms can report an accurate alignment result between two sequences. However, the accuracy is only achieved with a high computation complexity overhead. For very long sequences, the computation complexity means that the algorithm may not be able to be run on modern PCs or workstations. As an example, consider the human genome (which consists of about 3.2 billion bases). If the query sequence is 10,000 bases, 32 trillion scores are needed during the alignment process. Both the computation speed and memory consumption are beyond the capability of general-purpose computers. Thus, it is clear that the direct implementation of the dynamic programming-based algorithms is not feasible to solve the genome search problem. One alternative solution to handle this large volume of data is to use heuristics. However, this will affect the search sensitivity.

3.3 FASTA and BLAST

FASTA [102] is a software package for DNA and protein sequence alignment first presented in 1985, which plays a very important role in the development of sequence alignment tools. The heuristic applied by FASTA is simple: narrow the

search space before the time-consuming dynamic programming search. Although there are some differences between the DNA sequence search and the protein sequence search, FASTA can be divided into four steps:

- Identify regions of high similarity (see Figure 3.5 (a)). Identical substring pairs of length k, named k-tuples (normally, $k = 2$ is used for protein sequences, and $k = 4$ or 6 for nucleotide sequences), are detected between the query and database sequence.
- Rescan the score region using the score matrices leaving regions with high scores (see Figure 3.5 (b)).
- Combine high scoring sub-alignments into a single larger alignment, allowing the introduction of gaps into the alignment (see Figure 3.5 (c)).
- Use the banded SW algorithm [102] to calculate an optimal local alignment along the best matched regions (see Figure 3.5 (d)).

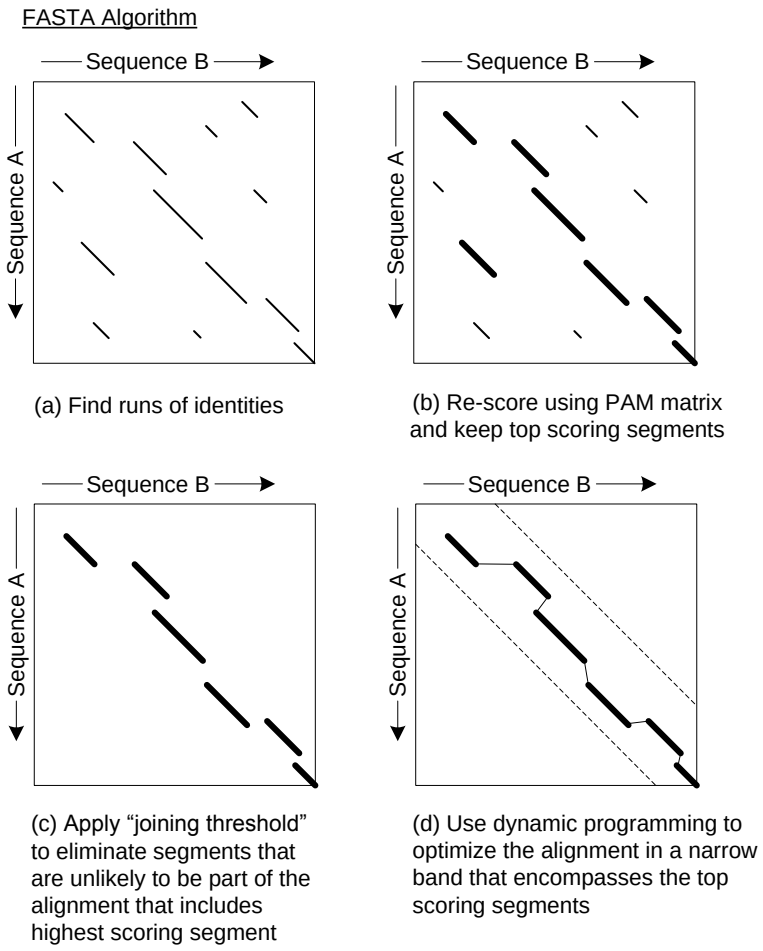


Figure 3.5 FASTA algorithm calculation steps [120]

The Basic Local Alignment Search Tool (BLAST), the dominant database search tool in molecular biology, is a set of algorithms designed to analyze sequences of different biological origins, such as the amino-acid sequences from different proteins or the nucleotides of DNA sequences. Now, cited more than 30,000 times, the original paper [7] is also one of the most frequently cited papers with significant influence, published in the 1990s. The family of BLAST programs was born in 1990 with the development of BLAST1, which is very fast and dedicated to search regions of local similarities without gaps. In 1996-1997, two new versions of BLAST (NCBI BLAST [91] and WU-BLAST [131]) which allowed for the insertion of gaps were released. NCBI BLAST is maintained by National Center for Biotechnology Information, and WU-BLAST is designed by Washington University in St. Louis. In this thesis, our study concentrates on NCBI BLASTN.

The NCBI BLAST package consists of a variety of programs suitable for specific applications and is listed in Table 3.1. In addition, the current standard distribution of NCBI BLAST also includes PSI-BLAST [8] to identify distant homologs and MegaBLAST [136] to rapidly compare highly similar nucleotide sequences.

Table 3.1 The BLAST family

Type	**Function**
BLASTN	Nucleotide query against a nucleotide database
BLASTP	Protein query against a protein database
BLASTX	Translated nucleotide query (6 frames) against a protein database
TBLASTN	Protein query against a translated nucleotide database (6 frames)
TBLASTX	Translated nucleotide query (6 frames) against a translated nucleotide database (6 frames)

The main idea of BLAST is that a statistically significant alignment usually contains high-scoring segment pairs (HSPs). BLAST uses a heuristic approach to approximate the computation-intensive SW algorithm to find the high scoring regions, which is less accurate than the original SW algorithm but can achieve a speedup of over 50 times. BLAST is computationally efficient and has a relatively good accuracy making it a popular choice for research in molecular biology. Similar to FASTA, BLAST also narrows the search space by finding local identity regions first (called seed generation). Then, it extends the seeds to get HSPs. Finally, it utilizes the SW algorithm to get the final alignments. Generally, the search process of BLAST seems similar to FASTA, as both of them conduct the search from local identities. However, the extension strategy is quite different for these two algorithms. In BLAST, individual seeds are extended in both directions to get a match alignment, whilst, in FASTA, individual seeds contained in the same diagonal are merged and concatenated using a banded SW algorithm. In general, BLAST can provide faster processing speed than FASTA when searching for significant patterns between the sequences with comparative sensitivity.

3.4 NCBI BLASTN profiling

BLASTN is composed of a pipeline of algorithms. With each stage of the pipeline, the associated algorithm becomes more computationally expensive and more sophisticated to identify biologically meaningful regions between the query sequence and the database sequence. The search space is gradually narrowed down by each pipeline stage. The stages, from the beginning to the last stage, are word matching, ungapped extension, and gapped extension. The rationale is that by reducing the amount of data (i.e. if irrelevant fractions are discarded as early as possible, the total computation time spent on the successive stages will be less). Figure 3.6 shows the data elimination process when data flows through the three pipeline stages of the NCBI BLASTN algorithm, where the highlighted part is the data flow. Past analysis on the various stages of the BLASTN pipeline reveals that the word matching stage is the most time consuming stage [60]. Thus, accelerating the computation of this stage will have the greatest effect on the overall performance.

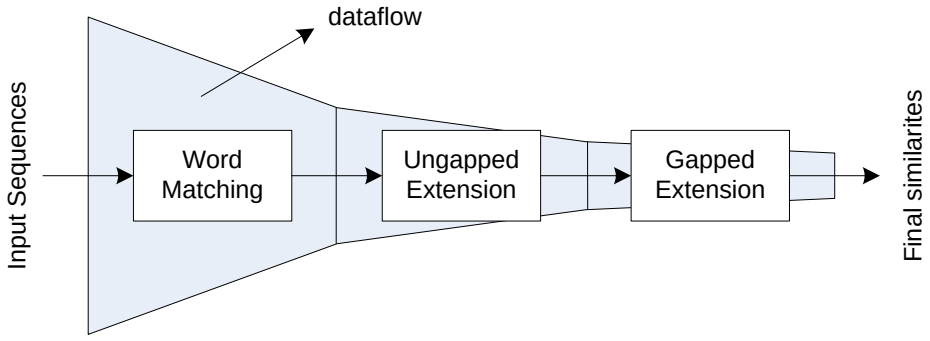


Figure 3.6 Pipeline stages of NCBI BLASTN algorithm

A *w*-mer is an exact substring match (of fixed length *w*) that occurs in both the query sequence and the database sequence or subject sequence. Normally, the default *w*-mer length is set to 11 for a nucleotide search. NCBI BLASTN speeds up the word matching stage computation by first examining *w*-mers on a byte boundary (i.e. an 8-mer). Figure 3.7 shows all possibilities that one 11-mer

contains 8-mer match. The solid boxes represent a matched 8-mer (two aligned bytes in memory). The dashed boxes represent all possible locations of an 11-mer relative to the two aligned bytes. The number inside the boxes represents the number of bases in the box. The word matching computation is done in two steps: (1) find all locations that have matched 8-mers, and (2) extend by three extra bases on both sides to get an 11-mer match. If two 11-mer occur in close proximity (for example, they overlap with each other), NCBI BLASTN implements a redundancy eliminator to avoid duplicate inspections in later stages.

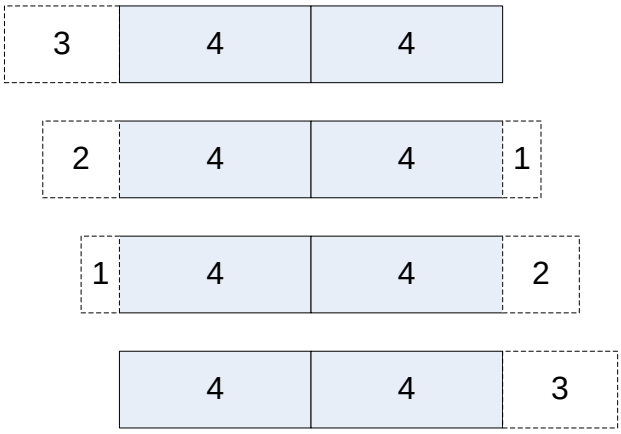


Figure 3.7 Possible 11-mer positions relative to byte boundary

The next stage is ungapped extension. NCBI BLASTN extends a *w*-mer into a HSP in two steps. The *w*-mer is extended back towards the beginning of the two sequences, then forward to their ends. A match character pair will receive a reward, while mismatch pair will get a penalty. A HSP's score in each direction is the maximum summation of all its rewards and penalties. If the final score is higher than the present threshold, it will be sent to last stage for further examination. The last stage is gapped extension, which has the greatest computation complexity. A dynamic programming algorithm is applied in this stage to get the detailed alignment score allowing gaps.

Even with these heuristic algorithms, BLAST searches still require a substantial amount of time due to the rapid growth of the genetic databases. To quantify the performance of NCBI BLASTN on a general-purpose CPU, an experiment is

conducted by [60] to measure the execution time consumed by the different stages of BLAST with the default parameter settings on a 2.8GHz Pentium 4 CPU, with an L2 cache size of 512KB and 1GB of RAM, running a Linux OS. The mouse genome (1.16 Gbase after removing repetitive sequences) is chosen as the target database sequence and queries of various lengths are selected randomly from the human genome. Table 3.2 records the percentage of time spent in each stage of NCBI BLASTN for various query sizes. A more detailed experiment is shown in Table 3.3 with respect to different query sizes. It is obvious that the word matching stage consumes over 80% overall computation time. Meanwhile, as the query size increases, the performance of word matching stage drops dramatically. This further leads to the overall throughput deterioration. For sequence alignment using a large database (billions of bases) and a long query sequence (millions of bases), it may take many hours even several days to finish the computation.

Table 3.2 Percentage of time spent in each stage of NCBI BLASTN [60]

Query size	**Stage1**	**Stage2**	**Stage3**
10Kbase	86.5%	13.3%	0.2%
100Kbase	83.3%	16.6%	0.1%
1Mbase	85.3%	14.65%	0.05%

Table 3.3 Performance of NCBI BLASTN software [60]

Query size	**Throughput**	**Stage1 (time / base)**	**Stage2 (time / match)**	**Stage3**
10Kbase	67.0	0.0129	0.231	71.3
100Kbase	8.76	0.0951	0.225	58.9
1Mbase	0.648	1.32	0.264	34.4
Units	Mbase/sec	μsec/base	μsec/match	μsec/align

Just as the databases are getting bigger and bigger, the usage of NCBI BLASTN is also growing. Currently, over 100,000 BLAST search requests are sent to NCBI each day. With the rapid-growing demands for conducting BLAST searches and

ever increasing large genomic databases, a faster version of BLAST can thus largely alleviate the current work burden. There have been several approaches to accelerate bio-sequence similarity searches. Some of these approaches use special hardware while others attempt to solve the problem in software. Hybrid approaches employ both general purpose computer and specialized hardware.

3.5 Previous work

3.5.1 Software programs for sequence alignment

When thinking about approaches to accelerate BLAST computation, the first thing that comes to mind would be utilizing software optimization techniques. In fact, many software implementations attempt to achieve this goal. MegaBlast [136] applies a greedy method, optimized for sequences with only slight differences, to accelerate the nucleotide sequence search process. During the alignment process, when a longer word size is applied, MegaBlast achieves up to 10 times speedup compared to other common sequence similarity search programs. MegaBlast is also capable of processing much longer DNA sequences than the conventional BLASTN implementation in an efficient manner. Compared to NCBI BLASTN, it provides a much faster runtime performance by sacrificing sensitivity.

The BLAST-like alignment tool (BLAT) [57], a software package for DNA and protein sequence analysis, is designed to speedup the search process between sequences with a high degree of similarity. BLAT first constructs an index of all non-overlapping *k*-mers in the genome sequence. The genome index is stored utilizing the inexpensive RAM space and can be reused for different search requests. With the help of the genome index, BLAT can quickly locate candidate regions (called hits) that are likely to have a high degree of similarity to the query sequence. Since it eliminates the need for repetitive scanning on the database, it achieves over an order of magnitude speedup comparing to BLASTN. As the number of candidate regions directly influences the search speed, BLAT applies a longer word length and restricts the hits to merely non-overlapping exact matches

to narrow the search space, which makes a trade-off between the processing speed and sensitivity.

The Sequence Search and Alignment by Hashing Algorithm (SSAHA) [95] aims to improve the search speed for large DNA databases (e.g. billions of bases). Several optimizations are added as a preprocessing stage. Sequences in the database are first split into consecutive *k*-tuples (contiguous bases of length *k*), and then used a hash table to store the position information for each occurrence of the *k*-tuple. When a search starts, all *k*-tuples in a query sequence are mapped to a given location of the hash table. If there is a hit, the position information will be extracted for further analysis. The computation efficiency is gained by the hash queries. SSAHA search is three to four orders of magnitude faster than BLAST, while its memory footprint is less than the suffix tree-based methods. Both SSAHA and BLAT require the offline indexation of the entire database before being used for searches. A trade-off exists among sensitivity, processing speed, and the storage space required for the indices. In fact, these techniques are less sensitive than the original algorithm in practice.

The PatternHunter series [84, 72] achieve both better runtime performance and sensitivity compared to BLASTN using different forms of pattern matching. They use a novel idea, the spaced seed model, to find all hits, which is based on the observation that hits at different positions of a region are not independent. The spaced seed model is designed to profit based on the following constraints: an appropriately chosen spaced model has a significantly higher probability to find at least one hit in a homologous region, while having a lower expected number of hits than the conventional consecutive seed model. If the spaced seed is not properly designed, there will be too many random hits, which will slow subsequent computations. In PatternHunter [84], a seed contains *k* discontinuous matched characters, where the relative positions of the *k* match locations are optimized in advance. PatternHunter II [72] implements the optimized multiple seeds scheme to further increase the sensitivity. PatternHunter reports a speedup of about 5-100 times comparing to BLASTN according to different data sizes with a similar

sensitivity. PatternHunter II is over a thousand times faster than the SW algorithm for DNA sequence searches at approximately the same sensitivity. However, finding the optimal spaced seed of given weight and length is NP-hard [72]. Meanwhile, the optimal seed models are specific to a particular dataset, i.e. repetitive seed model computations are required for different datasets.

The Diagonal Aggregating Search Heuristic (DASH) algorithm [59] works in three stages: (1) non-gapped alignments (diagonals) are identified with the help of the database indices; (2) these diagonals will be aggregated, if it is possible to conduct global alignment on the regions between them; (3) conduct dynamic programming-based alignment at the end of each aggregated diagonals. The query sequence filtering algorithm is special in DASH. If a *k*-tuple occurs excessively often, it will be excluded from the first stage and corresponding frequency information will be stored in each segment of the database. DASH reported a faster computational speed than BLAST by an order of magnitude for small datasets, with greatly improved sensitivity for a nucleotide search. However, the performance of DASH is not clear for longer query sequences.

MUMmer [32] is a new system for high resolution comparison of complete genome sequences. The system uses a combination of three ideas: suffix trees, the longest increasing subsequences (LIS) and Smith-Waterman alignment. The word matching stage design in MUMmer utilizes the suffix tree data structure which is more space efficient than the hashing strategy used in BLAST. As a useful attribute for the suffix tree structure, there is a unique path from the root to a leaf node to locate all suffixes within the suffix tree. Maximum unique matches (MUMs), which are used in later stages, are quickly identified with the help of the suffix tree and inspected by later stages. However, this system is built based on the assumption that the sequences are closely related, which limits its usage for finding alignments without exact matches.

The BLASTP program is a popular tool, widely used by biologists, to conduct searches against protein databases. Three optimizations are applied in [31] targeting the hit extension stage of the BLASTP algorithm since it occupies 93% of

the overall execution time. First of all, the data format of the query sequence and the code for scoring matrix indexation are changed. Secondly, the extension computation is conducted using two consecutive characters at a time (rather than one character applied in the original algorithm). Lastly, a forestall mechanism is applied to prefilter hits that are less likely to generate high scoring pairs, which can reduce the work burden for the more time-consuming extension stage. For each optimization, this work reports speedups of 15%, 48%, and 63%, respectively. However, the optimizations introduced also generate some loss in sensitivity.

As the fundamental process in many sequence analysis algorithms, approximate string matching against a large dataset plays an important role in Bioinformatics research. Biologists mainly use BLAST to locate candidate regions in DNA sequences, but BLAST is a heuristic approach which does not guarantee that all significant matches can be reported. An interesting auxiliary data structure is proposed in [25] to solve the approximate string matching problem with the help of the suffix array data structure. It maps the suffix array to the external memory, makes optimizations to improve the search efficiency and guarantees that all significant matches could be found. A quick lookup table (QLT) is applied such that binary searching on large intervals, needed for suffix tree simulation, is not required. This greatly improves the search efficiency of the suffix array. As this work uses a PC cluster to conduct computations, it provides two approaches to parallelize the computation, index partitioning (IP) and data partitioning (DP). Experiments showed that DP is a better choice than IP, if the error of the approximate string matching problem is large. Moreover, counting on the length of the DNA sequences, DP is probably more suitable than the IP strategy to conduct genome searches.

3.5.2 Heterogeneous systems for sequence alignment

Although software-based sequence aligners have already achieved a clear performance improvement compared to the original BLAST algorithm, they still face a great challenge brought by the ever growing databases. Structural

modification or custom hardware acceleration provides an alternative method to bypass obstacles that appear in the systems that rely on the general-purpose PC. Many designs to accelerate the sequence alignment process utilizing GPUs or FPGAs have appeared in the recent research literature.

Embedded in the NCBI-BLAST code, GPU-BLAST [128] performs protein alignments up to 4 times faster than the single-threaded NCBI-BLAST without compromising the accuracy of the produced alignments. As the comparisons of a query against the substrings of the database can be executed in parallel, GPU-BLAST develops a mechanism capable of distributing comparisons to different processors so that they were fully utilized and completed the assigned tasks at the same time. The parallelization efforts are mainly on the seeding and ungapped extensions steps, as these two steps consumed over 95% of the total execution time. Another key feature for GPU-BLAST lies in its data structure design. The location of each data structure in memory is carefully selected, depending on data size and the access frequency during execution.

CUDASW++ [79] is an open-source protein sequence database search algorithm using the SW algorithm, programmed in CUDA C++ for multiple GPUs. It employed a combination of inter-task parallelization and intra-task parallelization to achieve performance boost on the SW algorithm. Three techniques are applied to improve the design's performance: (1) a coalesced subject sequence arrangement improves the performance for multiple-thread subject sequence access, (2) a coalesced global memory access structure maximizes the bandwidth for intermediate data storage, and (3) a cell block division method is designed to reduce the burden on the requirement of the global memory. As the data structure is chosen based on the merits of the CUDA architecture, CUDASW++ achieves a very good performance for the SW sequence search.

FPGAs also have outstanding performance on parallel data processing and reconfiguration capability, which makes them a good option for algorithm acceleration. There are a number of designs that can speed up BLAST's performance using FPGA devices.

Tera-BLAST [123] is a commercial product designed by TimeLogic group. The Tera-BLAST design with a single J-series FPGA-accelerated similarity search engine achieves a speedup factor up to 434.2× over CPU-based NCBI BLAST+ and 161× over GPU-BLAST. In addition to the performance improvement, Tera-BLAST also provides more flexibility to support all flavors of BLAST. Unfortunately, the implementation details of Tera-BLAST are not in the public domain.

Another commercial product for BLAST-related computation is designed by SciEngines. This design accelerates the computation of the Smith-Waterman algorithm within the BLAST tool utilizing the RIVYERA FPGA-computer [112] which consists of 128 Xilinx Spartan3 FPGA chips. It achieves a 1600× speedup compared to a 3GHz single core desktop computer at a power consumption of 650 Watts. Similar to the Tera-BLAST, the detailed architecture of RIVYERA is unknown.

The conventional seed-based BLAST algorithm has been implemented on the RMEM architecture [66]: a combination of a FPGA board and a FLASH-NAND memory board. Both the seed and the neighborhood of each seed are indexed to accelerate the computation, which naturally increases the volume of the database index. For the human genome, the size of the index is about 45GB. The FLASH memory provides good support for this large volume of data. A systolic architecture is applied to compute the gap extension on the FPGA. It achieves over 10 times speedup for short query sequences. The indexation method applied in this design can be a performance bottleneck for larger databases, as the database is duplicated multiple times due to the simple neighborhood strategy and thus the index utilization efficiency can be low under the low match rate conditions.

RC-BLAST [88] is an early implementation of BLAST, which does not implement the entire application on the FPGA. Instead, it first profiles the application to identify the computation intensive segments. Then, it accelerates these critical segments using FPGA resources. This work also gives an illustration of the possible difficulties associated with the acceleration of this heuristic algorithm on

FPGAs. Although the final result is even slower than the software implementation, this is related to the FPGA technology used by the author.

TUC-BLAST [116] is an FPGA design provided by the Technical University of Crete (TUC) to accelerate DNA searches for small query sequences (1000 characters) regardless of the database size. The TUC architecture consists of N identical computing engines, each of which is a full implementation of all three pipeline stages of the algorithm. The hit finder and the extension unit are the basic processing components in this design. The hit finder stores a hash table of the query sequence using on-chip block RAMs (BRAMs) to speedup hit detection. The extension unit executes two comparisons in each cycle. It extends in both directions and compares the two pairs of letters. Due to its highly parallel structure, the TUC-BLAST implementation provides a significant performance improvement comparing to the BLAST program. However, for large query sequences, it is not wise to store the hash table using on-chip memory, as the BRAM resources are quite limited. In addition, it is not clear how to deal with hash collisions when constructing the hash table.

Two new versions of BLAST are presented in [49]: TreeBLAST and ServerBLAST. TreeBLAST is designed based on the idea that the alignment process can be viewed as a series of merging computations and a tree structure can be used to merge different segments in an efficient manner. One advantage of such a structure is that each node within the tree only requires a small amount of resources. Another advantage is that it can be easily pipelined level-by-level. In each cycle, ungapped alignment scores are generated without sensitivity loss. The ServerBLAST structure consists three parts: a systolic array that holds the query sequence, the alignment of interest (AOI), and k scoring servers. The database sequence is streamed through the systolic array and only the relatively infrequent AOIs are sent to the scoring severs for further processing, which also relies on the classic BLAST assumption of low seed rates.

The Washington University's Mercury system [60] is a reconfigurable architecture supporting disk-based computations with high speed data transfer interfaces. Many

aspects of both BLASTN and BLASTP are studied when deploying these programs on the Mercury system:

BLASTN on Mercury [60] presents several interesting ideas to accelerate the BLASTN word matching stage computation. The firmware system is decomposed into three sub-stages. The initial sub-stage implements a pre-filter using Bloom filters [17], which is an efficient method to quickly eliminate a large quantity of unrelated data with very small false positive errors; the middle sub-stage determines the corresponding query position of positive database *w*-mers (including both false positive and true positive answers) using a near-perfect hash table; the final sub-stage performs redundancy elimination which occurs when two matches have tail overlap. A detailed performance comparison between the Mercury firmware Stage1 implementation and NCBI BLASTN's software Stage1 implementation is provided. Speedups from 18 to 45 times (based on different query sizes) are reported.

Mercury BLASTP [54] is a hybrid architecture which decomposed the original operations into hardware parts and software parts. It is focused on accelerating the processing speed of the word matching stage for NCBI BLASTP. The BLASTP word matching stage is similar to that of BLASTN except for the use of two-hit modules due to the fact that protein sequences can generate more "matches". In order to generate alignment results with a similar sensitivity, it applies the same heuristics in hardware. The computation blocks are parallelized to achieve higher throughput rate. Special switch logic is designed to parallelize the hit generators and the two-hit modules. The effect of out-of-order, which is introduced by the parallel word matching modules, is closely examined and a related hardware design is also introduced to minimize this effect. The Mercury BLASTP implementation reports a speedup of 13 times over the software equivalent.

A parallel structure is proposed by [47] to accelerate the gapped alignment of the banded Smith-Waterman algorithm within BLASTP utilizing the Mercury system. Its performance is evaluated with a simulator. A performance improvement of over 54 times compared to that on a general purpose PC is reported.

Since the original ungapped extension algorithm in NCBI BLAST is not ideal for direct FPGA implementation, Mercury BLAST deploys a different, more FPGA-favorable ungapped extension algorithm which is suitable to be deployed as a pre-filter in front of NCBI BLAST stage2 [63]. The key features of this design are the fixed-size extension window and the systolic array based score computation scheme. The final design achieves over 20 times speedup against the BLASTN software due to its highly parallel and fine-grained pipelining architecture.

Another solution focused on accelerating the BLAST algorithm is presented by [130] using a new language called Macah. Macah is a C-like language created to break the barrier between software programmers and the hardware circuit design (i.e. accomplish FPGA designs for developers with little experience on HDLs but more familiar with the widely used C language). The proposed design is a mixture of Macah codes (implementing the word matching stage) and the original NCBI BLAST program (the rest of pipeline stages, ungapped extension and gapped extension). When compiling the code into Verilog HDL, the Macah compiler makes optimizations on concurrent operations. To reduce the repetitive off-chip memory access, a perfect or a near perfect hash table is usually required, which can lead to a large memory footprint to avoid hash collisions. In this design, cuckoo hashing [101] is used for the construction of the off-chip SRAM hash table, which alleviates the pressure on memory consumption without introducing any sensitivity loss. This makes it more competitive for systems with limited workspace memory. Unfortunately, this Macah design provides no performance comparison with that of NCBI BLAST.

Efficient dynamic programming algorithms exists for sequence comparison, but the significant scan time required becomes more and more severe as the database grows exponentially. A partitioning strategy is described in [97, 98] to implement bioinformatics database scanning using reconfigurable FPGA-based hardware platforms to gain high performance but with low cost. The SW algorithm is effectively mapped to fine-grained processing elements (PEs) resulting in an FPGA implementation that achieves about 170 times speedup for linear gap penalties and

125 times speedup for affine gap penalties, compared to a standard general-purpose processor. Strategies, which can further improve the system's performance using hyper-customization enabled by run-time reconfiguration, are also investigated.

Instead of following the conventional method using lookup tables, pipelined systolic arrays are utilized in [132] to find string matches. Two functions are accomplished by the proposed systolic array, seed detection and hit extension, which are the two most time-consuming pipeline stages in the NCBI BLAST algorithm. The multi-seed detection array consists of multiple processing elements, each of which holds one query character. When the database stream flows through each array, comparisons are conducted in parallel. Successive word-hits are combined into a valid seed and passed to the extension stage. In this paper, performance comparisons are made to different algorithms in the NCBI BLAST family (running with an Intel P4 CPU), such as BLASTP, TBLASTN, BLASTX, TBLASTX, and BLASTN. Speedups of 17, 48, 14, 71, and 10 are achieved for query sequences of 3062 bases, respectively.

To get speedups in the genetic sequence comparisons, the Smith-Waterman-Gotoh (SWG) algorithm [44] is implemented in [38] for sequence alignment on reconfigurable hardware. The SWG algorithm is a modification of the original SW algorithm to support gap penalties of a different form. The core of the implementation consists of a scalable pipeline, a controller and scratch-pad memory. The scratch-pad memory is introduced to support sequences larger than the pipeline length. The system achieves a speedup factor of 40 in a design that scales well with the size of the available hardware.

Bioinformatics algorithms are mapped to high performance reconfigurable computer (HPRC) systems in [2]. The influence of different degrees of parallelism on the design’s performance is evaluated. Two systems, the SRC-6 and the Cray-XD1, are selected as the test platform. To give comparable performance evaluation, both systems implement the original algorithm using a multi-node approach through the message passing interface (MPI). In this implementation, the query sequence is split and stored in each node, and the database sequence is broadcast to

all nodes for the similarity search. As expected, an order of magnitude speedup is achieved on both systems. The relationship between the number of nodes and the final performance is also evaluated: initially, the performances of both systems are linear to the number of nodes in the system; however, when the computation capability of the nodes is greater than the maximum communication capability, the performance drops as the communication overheads become the performance bottleneck of the whole design.

3.6 Summary

Genome sequence analysis, as an important research area in Bioinformatics, helps biological researchers understand the function of new genes and inspires the development of other related areas in genome research. Along with the rapid growth of the genomic databases, the need for an aligner with faster processing speed and reliable results always exists. Many tools for genomic sequence alignment have been designed over past decades either on general-purpose processors or utilizing heterogeneous platforms. When reviewing this rich literature, we find that seed-based heuristics is still an efficient approach to compute the degree of similarity even for a large database search. The performance improvements of past work are gained by utilizing better heuristic algorithms or optimizing different aspects of the existing heuristics. Heterogeneous designs take the outstanding computation capability of GPU or FPGA as an additional advantage to further optimize the target algorithm. Our FPGA accelerator design also belongs to the latter category: a more efficient parallel architecture to improve the performance of a seed-based algorithm (i.e. BLAST). The detailed architecture design is presented in the next chapter.

Chapter 4

Word Matching Accelerator for BLASTN

Based on the analysis given in previous chapter, the word matching stage is the most time-consuming part among the three stages of the original BLASTN algorithm. Therefore, accelerating the computation of this stage will have the greatest improvement on the overall performance. In this chapter, we propose a computationally efficient architecture to accelerate the data processing of the word matching stage utilizing FPGAs which are suitable candidate platforms for high performance computation due to their fine-grained parallelism and pipelining capabilities.

As discussed in Section 3.4, the basic idea underlying a BLASTN search is *filtration*. An efficient filtration method can largely reduce the computation time. Unlike previous designs merely focusing on the degree of parallelism, our proposed architecture also takes the actual data relationship between the query sequence and the database sequence into account. Two major contributions are made in our word matching stage architecture design: a novel parallel partitioned Bloom filter (PPBF) design and a block wise hash table data structure. As hash functions are well implemented in both of the architectures, a brief description of the hash function is included here.

4.1 Hash function and its applications

A simple explanation of a hash function is a set of mathematical functions that can map data within a large data scope into a small data scope. The general form of a hash representation is $x = h(k)$, where k is the key space, h is hash function, and x is the hash space. Hash functions are widely used for data indexing, context

compression, character comparison among large databases, duplicate record finding, web intrusion detection, encryption, finding similar strings in long DNA sequences, or many other application fields. Different hash functions will introduce a totally different performance evaluation, such as false positive probability, resource utilization, worst-case situation, and so on.

Although hashing is commonly used in computing applications, the performance of hash functions for specific applications is far from optimal. For example, if a design is targeted to FPGA devices, a hardware favorable hash function should be chosen for the implementation. Ramakrishna *et al*. [99] provide an analysis of the performance of hash functions and make a comparison between practical performances with theoretical performance predictions. Three practical hashing functions are chosen: the bit extraction hashing function, the hashing function from the XOR method, and the hashing functions from the class H_3 [22]. The results conclude that, hash functions that are chosen at random from the class H_3 achieve a performance consistent with the theoretical prediction, which is better than that of the bit extraction and the XOR hashing schemes.

The hash function for an off-chip hash table is more complicated than the ones used for Bloom filters. The main problem is related to hash collisions. A perfect hash function is one option to avoid hash collisions. An algorithm to construct minimal perfect hash functions (MPHF) is proposed in [18]. MPHF are hash functions generating no hash collision with minimum memory consumption, for a large set of keys utilizing external memory. The design of the algorithm considers four aspects: (1) the time required to construct the MPHF, (2) the space requirement for constructing the MPHF, (3) the computation complexity for the MPHF, and (4) the space consumption to represent the MPHFs. The proposed algorithm is simple and need just a small vector in the main memory to construct the MPHF. However, it is 34% slower than the fastest algorithm available to solve the same problem, which might impede its application when construction speed is the main concern.

A near-perfect hashing scheme is described in [20] for a dictionary of short bit strings, which is also used as part of the construction of Mercury BLAST. Near-perfect hashing is a heuristic variant of the well-known displacement hashing approach to build perfect hash functions. The core building block of the scheme is the hash functions from the H_3 family of full rank that guarantee uniformity of hashing. Two hash functions were chosen independently at random to further reduce the collision probability. Three aspects are taken into account for this design: (1) lookups in the dictionary usually required one probe; (2) the hash function should be simple enough to store on-chip and the dictionary should not require too much memory; and (3) the dictionary should be generated quickly. The experiments also demonstrate the performance of this novel hashing scheme.

Although a perfect hashing based algorithm can provide good performance for off-chip hash table access, the algorithm itself is likely to be quite complicated when the query sequence is fairly large. Cuckoo hashing [101] is a much simpler solution to avoid hash collisions. The main idea of this method is to use two hash functions instead of one, which provides two possible locations in the hash table for each key. The "kicking out" strategy also guaranteed that the hash table is collision-free. This algorithm has a constant worst-case bound on the time to do lookup. However, for a hardware implementation, each probe has to access two hash addresses in the off-chip memory, which means that, in the majority of cases, Cuckoo hashing will require higher memory access overheads than other perfect hashing queries.

The Bloom filter, a space-efficient probabilistic data structure conceived by [17] in 1970, is widely used to solve the membership examination problem for a large data set. False positive answers are possible, but false negative answers are not. New elements can be easily added to the set, but it is very difficult to remove them from the hashed results. If more elements are added to the date set, the probability of generating false positive answers will grow correspondingly. As the Bloom filter can quickly eliminate irrelevant elements and only has false positive answers, it becomes an ideal tool for the BLASTN word matching stage implementation.

Bloomier filters [23] utilize two tables for hash computation, rather than the hash bit-vectors used in Bloom filters. One table stores encoded values used for index computation for the second table; the other table is an associative array with effective range of *R*, the value of which can be modified dynamically. If the query data does not belong to the programming set, the function will return an invalid value with high probability (close to 1); otherwise, the return value is within the predetermined range *R*. This data structure is designed to reduce the lookup time and space consumption for false positive answers. There are many other alternatives for Bloom filters and Bloomier filters, but the core of these data structures is still the hash function.

The Bloom filter architecture has been used in a number of application fields besides bioinformatics. Various Bloom filter architectures have been proposed to achieve different functionalities. Multi-pattern matching algorithms for network intrusion detection systems (NIDS) are presented in [35] using multiple parallel Bloom filters on an FPGA, the performance of which is influenced by the number of patterns and the pattern length. The first algorithm performs the longest prefix matching (LPM) and the second algorithm is an extension of the first algorithm which handles long strings by combining with the Aho-Corasick algorithm [3]. The Bloom filter structures in these two algorithms effectively reduce the memory overhead and also accelerate the computation performance.

A string matching engine is proposed by [96] for high speed NIDS applications based on the Bloom filter architecture. The string matching engine consists of three major components, a Bloom filter accelerator, a dispatcher, and a parallel hashing (PH) engine. Unlike conventional Bloom filter designs mostly focused on reducing the false positive rates, this Bloom filter accelerator only uses a small number of hash functions and memory space to alleviate the work load on the PH engine. A dispatcher is designed to dispatch data and balance the work load for different channels of the PH engine. The PH engine is constructed based on parallel hashing and offered a string comparison phase to guarantees no false positive string matching.

A memory efficient and cache-optimized algorithm is presented in [87] for simultaneously searching for a large number of patterns in a very large corpus. Two special data structures are proposed to boost the algorithm's performance, the feed-forward Bloom filter (FFBF) and the cache-partitioned Bloom filter. FFBFs applies a second bit array to extend the functionality of the conventional Bloom filters and a different set of hash functions are used to determine the insertion location of the second bit array. As additional hash functions are applied for the search, the false positive rate can be largely reduced, which drastically reduces the memory for cleanup. The cache-partitioned Bloom filter data structure is an optimization specific for CPU implementation. The Bloom filter is split into two parts: the first part is smaller than the CPU cache available; and the second part is larger but accessed infrequently. The cache-partitioned Bloom filter is as effective as the classic Bloom filter, but has a much higher processing speed.

In network packet data stream, signatures need to be inspected under different application scenarios. [34] proposes a parallel bloom filter system to analyze stream data of various string lengths. It utilizes the multi-port embedded memories to support multiple concurrent memory accesses and the m-bit vector is divided into smaller segments, which will increase the false positive probability correspondingly. The proposed system is implemented on a Xilinx FPGA supporting 10,000 strings operating at 81 MHz. It also provides an analysis on the trade-offs between the memory resources and system performance, which is important for bloom filter architecture design.

PERG [50] is a FPGA-based system created to accelerate the pattern search for a stream against the large fixed string database. It consists of three major components: inspection unit, metadata unit, and fragment reassembly unit. Within the inspection unit, multiple Bloomier filter units (BFU) are constructed to filter irrelevant data. Two hash functions are applied for each BFU based on the shift-add-xor (SAX) [107] method. CRC checksums for each segment are stored in the BFU to reduce false positives. PERG can fit over 80,000 patterns into a single

FPGA chip, which achieves a 26× improvement in terms of memory density over other NIDS pattern match engine.

The previous work on accelerating the hash function and its applications (described above) indicates that it may be possible to accelerate the NCBI word matching algorithm with the help of efficient hash implementations. The next section examines this possibility.

4.2 Word matching stage acceleration

The first stage of BLASTN is used to find "seeds" or word matches. A word match is a common substring of length *w* (referred to as a "*w*-mer") that appears in both the query sequence and the database sequence. Using the alphabet {*A*, *C*, *G*, *T*}, NCBI BLASTN reduces storage and I/O bandwidth by storing the database using only two bits per letter (or base). The default *w*-mer length for a nucleotide search is set to 11. The word-matching stage implementation of NCBI BLASTN first examines *w*-mers on a byte boundary (i.e. 8-mers) and then extends 8-mer matches in both directions to find possible 11-mer matches. If two matching 11-mers occur in close proximity, they are likely to generate the same HSPs. NCBI BLASTN therefore implements a redundancy eliminator to avoid repetitive inspections on the same segment in later stages.

Our FPGA-based accelerator design for BLASTN does not follow exactly the same working mechanism presented in the NCBI BLASTN software. Instead, we have chosen FPGA favorable algorithms to achieve the same functionality. Our word-matching stage design can be decomposed into three sub-stages, as shown in Figure 4.1. The first sub-stage is a parallel Bloom filter; the second sub-stage is a false positive eliminator to examine the data passed the parallel Bloom filters, and the last sub-stage eliminates redundant matches. The sub-stage composition is similar to that of Mercury [60], but the detailed architecture is very different.

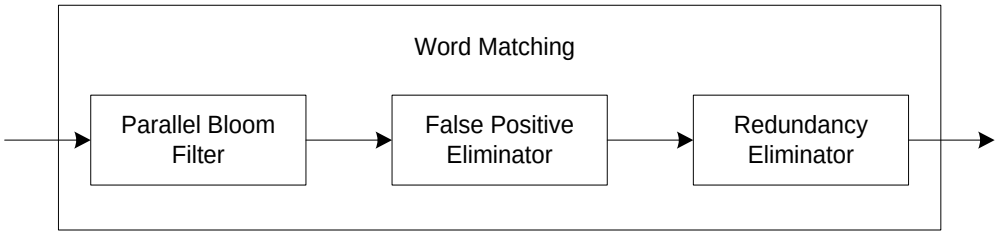


Figure 4.1 Three sub-stages of our FPGA-based accelerator for the word matching stage of BLASTN

4.3 Parallel Bloom filter architecture

The word matching stage aims to find good alignments containing short exact matches between a query sequence and a database sequence. Such matches could be computed using data structures such as hash tables or suffix trees. One drawback for the hash table or the suffix tree solution is that the memory footprint can be very large for long sequences. An alternative solution to this filtration problem is to use a Bloom filter, which is a simple space-efficient randomized hashing data structure suitable for FPGA implementation.

Bloom filters represent a set of given keys in a bit-vector. The provided hash functions support the insertion of keys (in the programming stage) or the search of keys (in the querying stage). The Bloom filter is a space-efficient data structure for membership test allowing a small portion of false positive answers. Since later stages can efficiently eliminate the false positive matches, the space savings far outweigh the introduction of false positive answers.

A brief description of the Bloom filter's definition, working processes and false positive probability is presented below:

Definition: a Bloom filter is defined by a bit-vector of length m, denoted as $BF[1,..., m]$. A set of k hash functions h_i: $K \rightarrow A$, $1 \leq i \leq k$, is associated to the Bloom filter, where K is the key space and $A = \{1, \dots, m\}$ is the address space.

Programming: for a given set I of n keys, $I = \{x_1, \dots, x_n\}$, $I \subseteq K$, the Bloom filter's programming stage is described as follows. First, the bit vector is initialized

with all zeros; then, for each key $x_j \in I$, the k hash values $h_i(x_j), 1 \leq i \leq k$, are computed; subsequently, set the bit vector to one according to the k hash values (i.e. $BF[h_i(x_j)] := 1$ for all $1 \leq i \leq k$).

Querying: the querying stage of a Bloom filter works the same as its programming stage. For a given key $x \in K$, compute k hash values $h_i(x), 1 \leq i \leq k$. If any of the k queries $BF[h_i(x)]$, $1 \leq i \leq k$, is zero, then the membership test is failed $x \notin I$. Otherwise, x is treated as a member of I with a high probability.

False Positive Probability: the appearance of false positive answers is due to the fact that the k bits in the bit-vector can be set to one by any of the n keys (i.e. different keys can be hashed to the same location). However, we should also notice that a Bloom filter only produces false positives but never false negative answers to the query. The false positive probability of a Bloom filter (denoted as *FPP*) is given by [17]:

$$FPP = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{4.1}$$

It is clear that *FPP* is determined by three parameters, the bit-vector length m, the number of keys n, and the number of hash functions k. *FPP* decreases as the bit-vector size m increases, and increases as the number of keys n increases. It can be shown that for a given m and n, the optimal number of hash functions k_{opt} is given by:

$$k_{opt} = \frac{m}{n} ln2 \tag{4.2}$$

The corresponding *FPP* is then:

$$\left(\frac{1}{2}\right)^{k_{opt}} \approx 0.6185^{\frac{m}{n}} \tag{4.3}$$

If we maintain a fixed *FPP*, the bit-vector size needs to scale linearly with the inserted key set. Table 3.4 shows the required Bloom filter size for *FPP* values of around 10^{-5} and around 10^{-3}, and varying query lengths. About 2Mbit of memory is enough for scanning a database with a query sequence of length 100kbase and 16 hash functions.

Table 4.1 Bloom filter memory size (in bits) for different parameter combinations

Query size (n)	**# of hash functions (k)**	**Bit vector length (m)**	**FPP**
1k	16	23,084	1.53e-5
10k		230,832	
100k		2,308,313	
1k	8	11,542	2.91e-3
10k		115,416	
100k		1,154,157	

The conventional architecture (i.e. the structure based on the definition of a bloom filter) for the identification of *w*-mers using a Bloom filter is shown in Figure 4.2. The Bloom filter has been programmed by parsing the query sequence into overlapping substrings of length *w* in the preprocessing step. One important observation for the conventional architecture is that the memory containing bit-vector must support *k* random lookups for the same query *w*-mer, which poses a question for the FPGA designers - how to efficiently implement multiple hash queries?

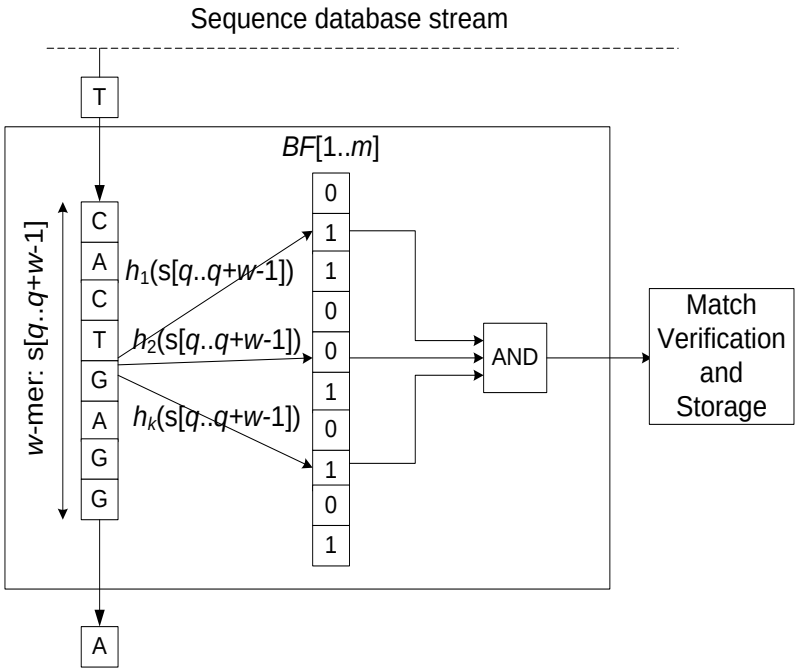


Figure 4.2 The conventional architecture for identifying *w*-mers in a database stream using a Bloom filter

Although the conventional Bloom filter architecture is efficient for the member ship test, its direct implementation is not suitable for a high performance design on an FPGA. Current embedded memories can only provide limited access during the same clock cycle (2 queries at the same clock cycle). Pipelining or memory duplication is required for multiple memory access requests. For example, Mercury implements the Bloom filters using the conventional structure. It doubles the clock frequency of the BRAMs to reduce memory consumption, but still requires four copies of the m-bit vector to process 16 hash queries at the same clock cycle. The limited on-chip memory becomes one bottleneck for the use of the Bloom filter.

Our design takes advantages that mismatches appear far more frequently than matches in the BLASTN word matching stage and a match key has much tighter requirements than mismatches (once a key fails in any of the *k* hash queries, it will be defined as a mismatch; in contrast, a match key must pass all hash queries). The computation efficiency will be compromised, if a single key is sent to all hash functions for membership testing, especially under low match rate conditions. Thus,

our idea is to divide the *k* hash functions into different groups, with each group used for a different hash query. We apply three techniques to improve the throughput compared to the conventional Bloom filter architecture.

- ***Partitioning***: we first partition the Bloom filter vector into a number of smaller vectors, which are then queried by independent hash functions.
- ***Pipelining***: we further increase the throughput of our design using a new pipelining technique.
- ***Local stalling***: we use a local stalling mechanism to guarantee all *w*-mers are tested by the Bloom filter.

In our design, we partition a Bloom filter of bit-length *m* into *k* embedded memory blocks (BRAMs) of length *m/k* as shown in Figure 4.3, where *k* is the number of hash functions. Each BRAM block is an independent and identical memory space. The hash functions applied randomly map the input keys into the range [0, *m/k*) of the localized BRAM. As a different set of hash functions are chosen, the partitioned false positive probability changes to:

$$FPP_p = \left(1 - \left(1 - \frac{1}{m/k}\right)^n\right)^k \tag{4.4}$$

For instance, using the parameters mentioned in Table 4.1, we will get $|FPP_p – FPP| \leq 5.5e\text{-}8$ for $k = 16$ and $|FPP_p – FPP| \leq 5.5e\text{-}20$ for $k = 8$. Thus, the negative influence on the *FPP* introduced by the Bloom filter partitioning is negligible.

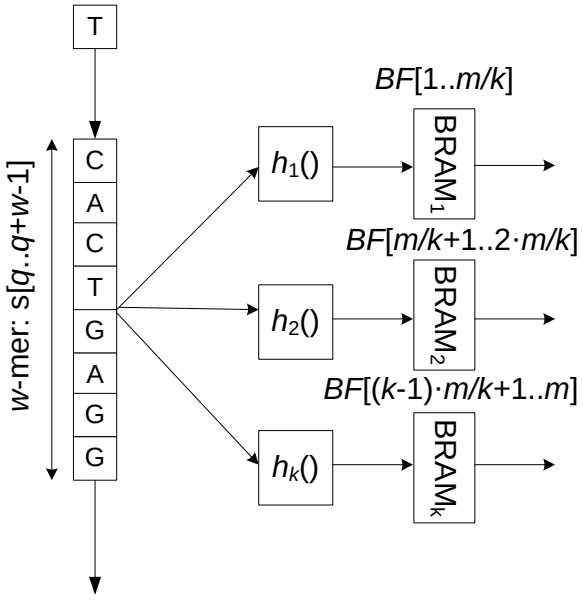


Figure 4.3 Partitioned Bloom filter architecture using on-chip BRAM modules

In order to further improve the Bloom filter's throughput while keep the BRAM space consumption within a reasonable limit, we first present a space-saving pipelined partitioned Bloom filter with *k/P* hash functions, denoted as PPBF(*k/P*), where *P* is the degree of parallelism. Figure 4.4 shows one structure of the pipelined partitioned Bloom filter. Pipeline registers can be inserted along the bit-AND path. In each clock cycle, it can support *k/P* different hash queries.

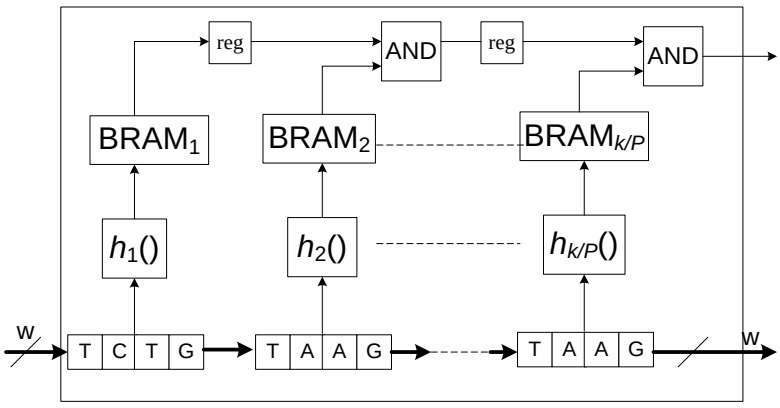


Figure 4.4 The PPBF architecture with *k/P* hash functions

The hash functions used in the PPBF block are chosen from the H_3 family [106], which can be efficiently implemented in hardware. An H_3 hash function has the

following form: suppose the input data stream X with b bits is presented in binary format as $X = <x_1, x_2, ..., x_b>$; we calculate the i^{th} hash function over X, $h_i(X)$ as:

$$h_i(X) = (d_{i1} \cdot x_1) \oplus (d_{i2} \cdot x_2) \oplus ... \oplus (d_{ib} \cdot x_b) \quad (4.5)$$

where "$\oplus$" is a bitwise XOR operator and "$\cdot$" is a bitwise AND operator; d_i are predetermined random numbers in the range [0, ..., m-1]. Both the AND and XOR operations can be implemented in parallel to shorten the computation on FPGA. In addition, the total number of hash functions required is small. Thus, the resource needed for the hash function representation is negligible. Although hash collision might appear in both the programming and querying stage, the hash table lookup stage can eliminate all negative influences.

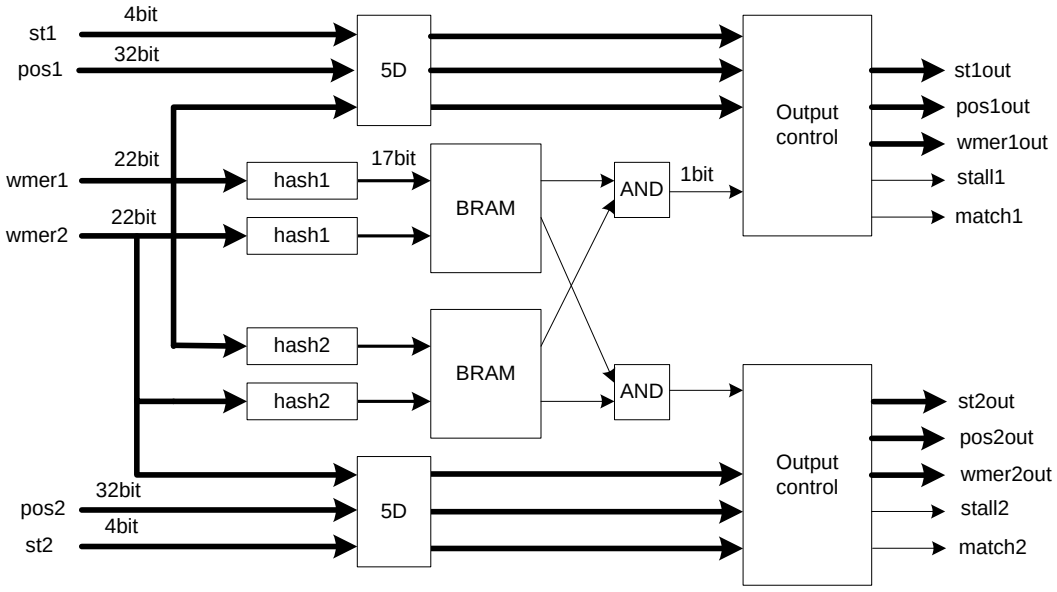


Figure 4.5 PPBF architecture with two hash functions

The implementation of the PPBF architecture, with two different hash functions, is shown in Figure 4.5. Two sets of input signals are designed as the input ports: the query 11-mers (*wmer1* and *wmer2*, each 22bits wide, representing 11 DNA characters), the position information (*pos1* and *pos2*, each 32bits wide and thus supporting databases of up to 4 billion characters), and the state information (st1 and st2, each 4bits wide, recording the number of PPBFs already passed). The hash function implementation follows equation 4.5, which maps each 22bit *w*-mer to the

hash space. The BRAM size is 128Kbit representing the "*m/k*-bit" vector, which is tunable for different FPP computations. The *output control* block monitors the status of each input *w*-mer and outputs only successful hash queries. The match signal indicates a true match *w*-mer and the *stall* signal is used to control the data flow of the next PPBF. Pipeline registers are inserted into hash function, BRAM and output control modules to improve throughput and introduce a 6 cycle latency.

The second step is to implement *P* PPBFs in parallel. Figure 4.6 is an example containing eight PPBFs, where each PPBF is implemented using the architecture shown in Figure 4.5. This implementation reduces the FPP, making it more computationally efficient. We refer to the design in Figure 4.6 as an 8×2 parallel Bloom filter (denoted as PBF8×2). When we use dual port RAM for the "*m/k*-bit" vector module, each partitioned memory block can support an additional hash query at the same clock cycle. Thus, the system's throughput can be doubled. Assuming our design works with a 100MHz clock, it can thus achieve a theoretical maximum throughput of 1.6Gbases/s under the zero-matching condition (i.e. all querying *w*-mers fail at the first PPBF they met).

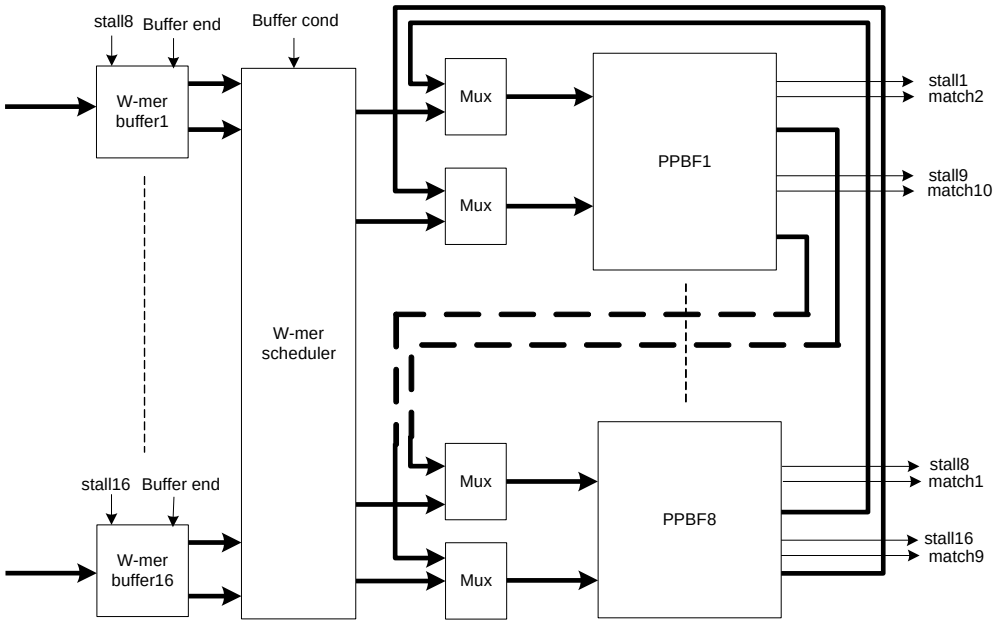


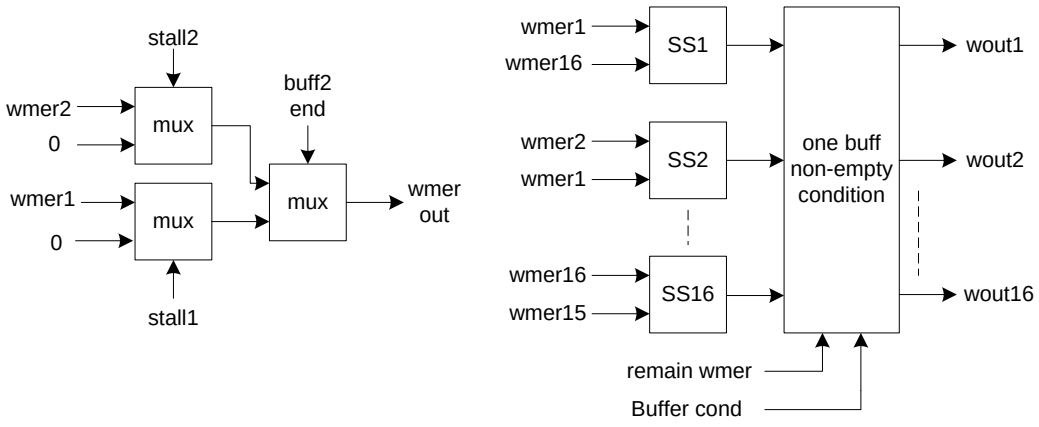
Figure 4.6 Parallel Bloom filter architecture consisting of eight PPBFs with two hash functions each

As shown in Figure 4.6, a *w*-mer passing one PPBF will be sent to the next PPBF for further examination to reduce the FPP, which would introduce a data collision between the match *w*-mer and the incoming new *w*-mer. To avoid such a collision, we apply a local stall mechanism as: if the output match bit of $PPBF_i$ equals zero for any $1 \leq i \leq P$, indicating that a mismatch is found, then a new *w*-mer is input to the PPBF; if the output match bit of any $PPBF_i$ equals one, the corresponding *w*-mer is passed to $PPBF_{(i+1)}$ for all $1 \leq i \leq P$-1; at the same time, a corresponding input *w*-mer from the *w*-mer buffer should be stalled, based on a local stall signal, to prevent a *w*-mer loss. The match output from $PPBF_P$ is sent to $PPBF_1$ for further testing. A true match is generated if and only if a *w*-mer passes all PPBFs. The local stall signal is connected with *w*-mer buffers. Once the *stall* signal is high, corresponding buffer will stop popping data.

Although the local stalling mechanism is applied to guarantee that there is no *w*-mer loss, a processing speed difference will still appear among different *w*-mer buffers. This is because matches are not equally distributed in the database sequence. Some of the *w*-mer buffers would require a longer period to pop all *w*-mers. Therefore, we also integrate a *w*-mer scheduler to control the *w*-mer flow. The rule for the *w*-mer buffer scheduling is:

If $buff_i$ is not empty
    If $buff_{i+1}$ is not empty
        pop *w*-mer normally when there is no *stall*
    else
        If there is no *stall* for all buffs
            If only $buff_i$ is non-empty
                pop *w*-mer to all PPBFs
            else
                pop *w*-mer normally
        else
            pop *w*-mer to PPBF responsible for $buff_{i+1}$
else
    If all buffs are empty
        load new *w*-mers to all buffs

Buffer cond is a control signal that informs the w-mer scheduler about the empty buffers. The hardware implementation of the scheduler is shown in Figure 4.7 (b). The scheduler consists of 16 sub-schedulers (SS) to control the data flow for each *w*-mer buffer. The SS module (shown in Figure 4.7 (a)) is the switch logic to determine the output *w*-mer. To reduce the latency for our PBF design, the scheduler is constructed using pure combinational logic.



(a) scheduler small (SS) (b) scheduler structure

Figure 4.7 (a) sub-scheduler and (b) scheduler architecture

4.4 False positive eliminator design

The second sub-stage of our word matching accelerator design is false positive elimination, which includes two objectives:

- Eliminate all false positive matches generated by the Bloom filters.
- Get the corresponding position information in the query sequence for true positive *w*-mers.

One solution for this sub-stage design is to use a hash lookup table. The position information of each *w*-mer from the query sequence is stored in the hash table. A hash table with one million entries storing position information for a 100kbase query sequence requires at least 17Mbit of memory space (17 bits are needed to represent 100k positions). It is clear that the memory required is significantly greater than that provided by the on-chip BRAMs (for our target FPGA chip, it only provides about 10Mbit BRAM resources). Thus, we have to store the hash table in an external SDRAM attached to the FPGA.

Hash collision and duplicate keys are two common problems for simple hashing strategies. The former will hash two different queries to the same location, while the latter may miss additional position information. Both of them require extra access to the off-chip SDRAM to get the correct data. As off-chip memory access is much slower than the on-chip memory access, this can be a potential performance bottleneck for the word matching stage design.

In previously reported designs, a perfect hash function [18] has been applied to construct the hash table. A perfect hash function for a set of n keys maps each key to a distinct table entry with no collisions among the keys in the set. However, a perfect function is not easy to generate, especially when the key set *n* is large. In addition, the representation of the perfect hash function usually needs a significant amount of FPGA resources and may compete with the Bloom filter design. The Mercury BLASTN design [57] implements the hash table using a near-perfect hashing strategy, which bypasses the constraint for a perfect hash function.

However, considerable effort is still required to get the "near-perfect" hash functions. Cuckoo hashing [101] is another effective hash strategy used to avoid hash collision, where two independent hash functions are used for a single hash query. However, additional off-chip hash table access may reduce the overall performance, and, it still cannot eliminate hash collisions for duplicate keys. In our design, we try a less complicated approach with few hash collisions, called the bucket hash.

Our idea is simple: although it is difficult to find a perfect hash function for all *n* keys, it might be easier to find a perfect hash function (or a hash function with only limited collisions) for a subset of keys, if the size of the subset is small enough. Bucket hashing works as follows:

- Firstly, sort the query w-mers into different buckets according to their prefix (if the prefix length is properly chosen, the number of *w*-mers in each bucket is relatively small).
- Secondly, find a simple hash function that is collision-free for all *w*-mers in the same bucket. If it is not possible to find such a perfect hash function, uses the hash function with the minimum number of hash collisions.
- Finally, construct a quick lookup table (QLT) which stores the "collision-free" hash functions for each bucket.

An example of the off-chip hash table construction using the bucket hashing strategy is shown in Figure 4.8. Query *w*-mers sharing the same prefix are programmed to a given area in the hash table without hash collision. For example, to support a 10kbase query sequence, we construct a QLT of size 24kbit with 4k entries. If a longer query sequence is applied, the number of *w*-mers in the same buckets will increase correspondingly. This puts a greater burden on finding a perfect hash function. In rare cases, it is difficult to get a perfect hash function with a simple representation. Under such circumstances, a *secondary table* is applied to store *w*-mers that would introduce a collision. A collision flag "C" is set to identify the colliding keys. As the total number of such *w*-mers is small, it only has a minor influence on the overall performance. Although bucket hashing can avoid hash

collisions for the majority of keys, an additional operation is still required for duplicate keys.

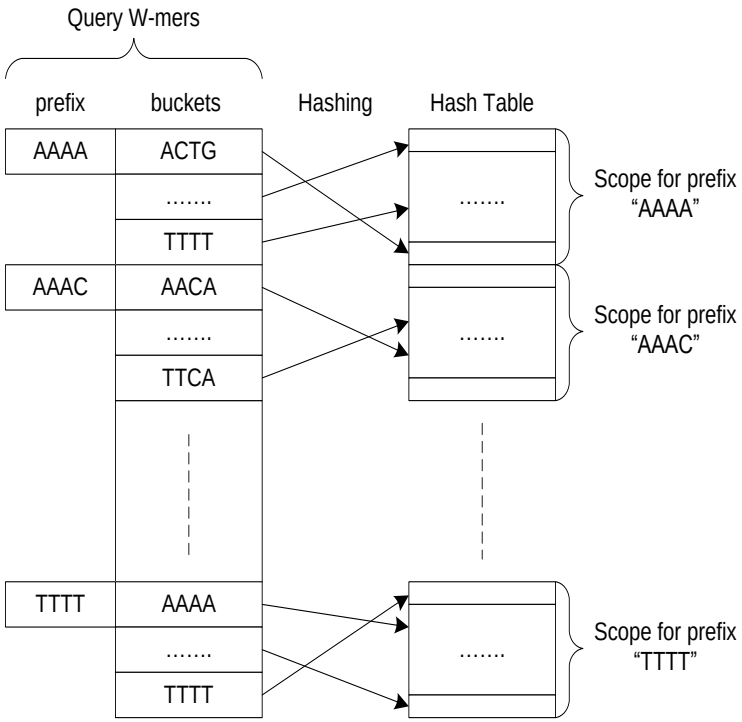


Figure 4.8 Hash table construction using bucket hashing

As duplicate keys cannot be distinguished by hash functions, we construct a separate table, called a *duplicate table*, to store all duplicate keys. We also set a duplicate flag "D" field to indicate the appearance of a duplicate key. Each element in the primary table, the secondary table, and the duplicate table takes 64 bits. Figure 4.9 shows the bucket hash data path. The definitions for the primary, secondary, and duplicate table fields are:

En	effective slot in the primary table, i.e., query *w*-mer was programmed into this slot during the programming stage
D	duplicate flag
C	collision flag
wmer	query *w*-mer
PTR	if the collision flag is "1", then it indicates a position in the

secondary table; if the duplicate flag is "1", then it indicates the address in the duplicate table; otherwise, it indicates the corresponding address in the query sequence

End flag that indicates no more duplicates or collisions for the current *w*-mer

XX unused bits (set to 0)

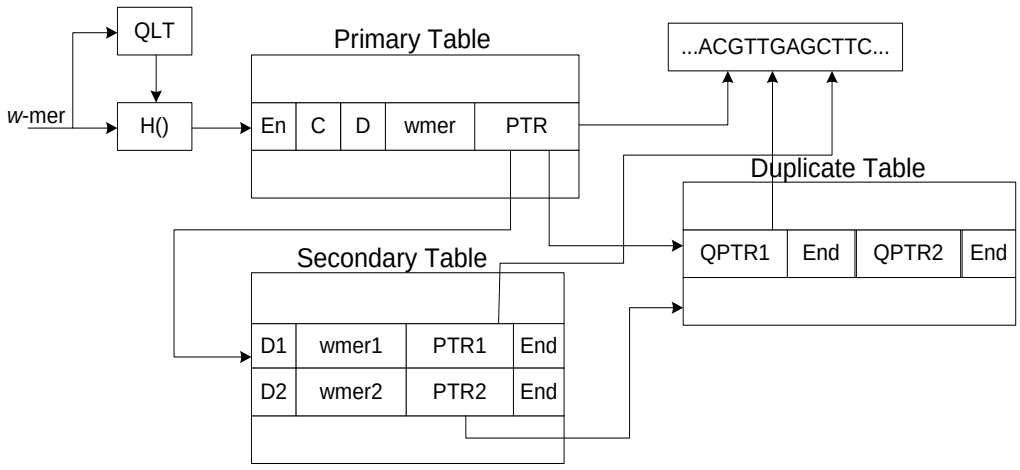


Figure 4.9 Bucket hash data path

There are several different cases for the hash table access. If a *w*-mer *x* maps to the primary table entry *h*(*x*) without collision and duplication, it only requires a single off-chip memory access. In the duplication-only case, the address we get from the primary table indicates a start location in the duplicate table; we should read out all the position information from this location until an "*End*" flag is met. If the collision flag is set in the primary table, we should access the secondary table based on the *PTR* recorded in the primary table; if duplication appears at the same time, the collision table access has a higher priority (i.e. first access the secondary table using *PTR* from the primary table; then, access the duplicate table using *PTR1* or *PTR2* and retrieve the correct duplicate position *QPTR1* and *QPTR2*). The control logic of the hash table lookup is illustrated in Figure 4.10. The query *w*-mers are also stored in the primary table and the secondary table. Each time we conduct a hash lookup, we will also compare the *w*-mer from the database against

the one from the off-chip table. Thus, we can guarantee that there is no false positive match after the hash table lookup stage.

```
address ← h(x)                          // address for primary table
if ((C = 0) and (D = 1))                // only duplication appear
    address ← PTR + DupOffset           // address for duplicate table
    return QPTRs                        // return QPTRs until meet an END
else if (C = 1)                         // collision apepar
    if (D = 0)                          // no duplication jump to secondary table
        address ← PTR + SecOffset
        return PTRs                     // return PTRs until meet an END
    else                                // duplication appear jump to duplicate table
        address ← PTR + DupOffset
        return QPTRs
```

Figure 4.10 Hash table control logic

4.5 Redundancy eliminator design

In order to prevent generating the same sequence alignment repetitively during the ungapped extension stage of BLASN, a redundancy filter is applied to eliminate *w*-mers that lead to the same ungapped extension stage. Each matching *w*-mer can be represented using its location pair (q_i, d_j), where q_i and d_j are the locations of the query and database sequence, respectively. The diagonal of this *w*-mer is defined as $D = q_i - d_j$. Redundancy matches are eliminated by examining their diagonals. In NCBI BLASTN, it also uses the feedback from the ungapped extension stage to eliminate redundancy matches. In contrast, our design is less stringent. We only eliminate "true overlapping" match *w*-mers, i.e., if two consecutive matches share the same diagonal and they have an overlapping part, we discard the latter as a redundant match. The non-overlapping diagonal will be updated, once a non-overlapping match is found. An example of the redundancy eliminator is shown in Figure 4.11. For example, match alignment *a* and alignment *b* locate on the same diagonal and they are overlapped to each other. Thus, alignment *b* is a redundant match. In contrast, a gap exists between alignment *c* and *d*. Both of them can pass

the redundant filter. Although our heuristic is less stringent than NCBI BLASTN's, there is no significant influence on the overall performance.

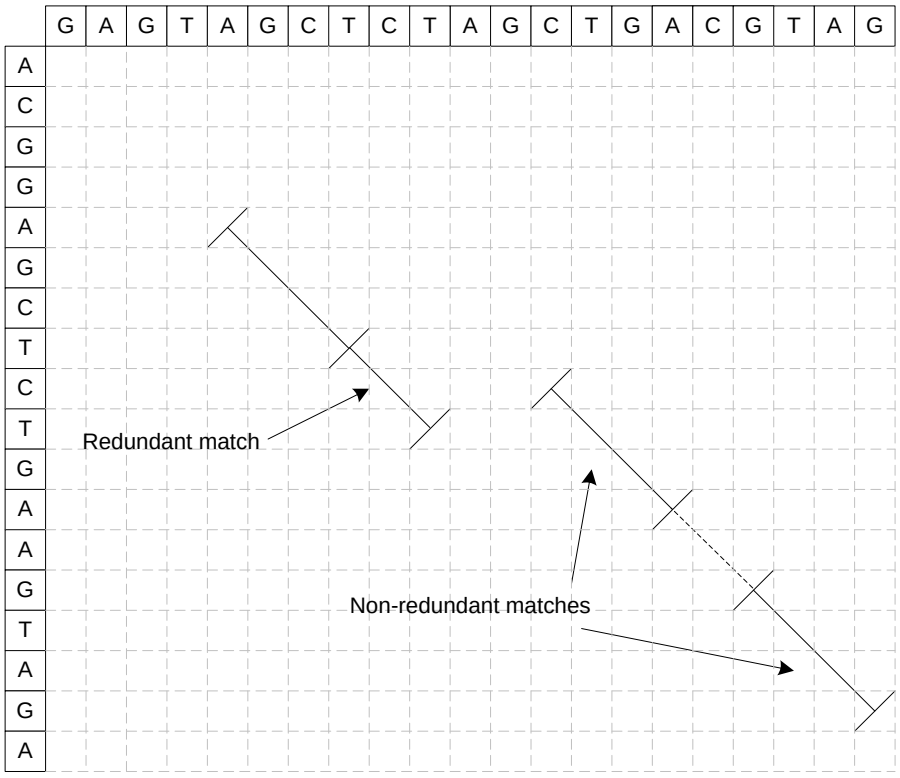


Figure 4.11 Redundancy filter example

4.6 Performance analysis

The word-matching stage accelerator has been implemented using Verilog HDL and integrated into the DRC coprocessor system [34]. In the DRC system, a Xilinx Virtex-5 LX330 FPGA chip is available for the user application. A large volume of data can be store using the DRC system's off-chip memory, which consists of up to 8GB of DDR2 SDRAM with a maximum bandwidth 3.2GB/s and 512MB of low latency RAM with a maximum bandwidth of 1.4GB/s. In each clock cycle, the parallel Bloom filter can receive up to 16 new *w*-mers to do the membership examination from local buffers. Table 4.2 records the resource utilization for our complete word matching design.

Table 4.2 Resource utilization

	Resource used
Number of Slice Registers	58,290
Number of Slice LUTs	44,111
Number of bonded IOBs	903
Total Memory used (KB)	6,606

The parameters used for the Bloom filter design are: *m* is set to 2,097,152 (2Mbit for m-bit vector); *k* is set to 16 (16 different hash functions); *n* is a variable depending on the query sequence length (we have tested different query sequences ranging from 10kbase to 100kbase in a single scan). In order to quantify the performance improvements of our word matching accelerator, we have designed several tests to simulate possible large-scale DNA sequence comparisons.

4.6.1 Performance comparison against NCBI BLASTN Stage1

We have compared our FPGA implementation to NCBI BLASTN version 2.2.9 [91], running on a general-purpose workstation containing an Intel Core2 Duo CPU (3.2GHz) with 8GB RAM. Both the Virtex-5 LX330 and the Core2 CPU were released in 2006 using 65nm technology. As both platforms belong to the same technology generation, we think the performance comparison between them is a proper choice. The parameters for both the FPGA and the software program are set to the same default values. We have chosen the mouse genome (mm8, consisting of about 1.4Gbases after removing unknown characters and Ns, including 19 sequences) as the database sequence. Query sequences of different sizes have been randomly chosen from the human genome (hg18, 10kbase, 50kbase, and 100kbase). The maximum clock frequency that our FPGA design can support is 133.68MHz (reported by the Xilinx timing analyzer after place and route). Thus in our experiment, we have tested our design with a 133MHz clock

frequency. Table 4.3 shows the sensitivity comparison between NCBI BLASTN Stage1 and our word matching accelerator. The database sequences are processed sequence by sequence.

Table 4.3 Sensitivity comparison between NCBI BLASN and DRC accelerator

Query	**10kbase**	**50kbase**	**100kbase**
# of hits (FPGA)	9,027,656	39,686,463	82,305,804
# of hits (NCBI Stage1)	5,223,591	29,364,313	61,974,681
# of HSPs (FPGA*)	609	850	1270
# of HSPs (NCBI Stage2)	609	827	1212
# of matches (FPGA*)	609	850	1270
# of matches (NCBI Stage3)	609	800	1183

* The number of HSPs and matches are computed using a software program with the same parameter settings as the NCBI BLASTN.

Our design reports more 11-mer hits compared to the NCBI BLASTN Stage1. In our word matching accelerator, the Bloom filter only introduces false positive results with no false negatives, which guarantees no sensitivity loss for the true matches. The off-chip hash table structure, which covers all situations (collision hits and duplicate hits can be examined by looking up the secondary table and the duplicate table, correspondingly), eliminates all the false positives from the Bloom filter stage. Thus, our FPGA design can report all 11-mer hits between the query sequence and the database sequence. In fact, the NCBI software might report fewer hits due to its optimizations on accelerating the word matching search process. For example, for a 100kbase query sequence, the software first scans the database sequence for 10-mer matches with a step length of two. Then, extends one extra base in both sides to get a 11-mer match with the search terminating once a match is found, if a long *k*-mer ($k > 11$) match exists, 11-mer hit losses will appear. The lost hits are expected to be re-examined by the ungapped extension stage which extends hits base-by-base in both directions. However, sensitivity loss might appear due to the optimizations applied in the word matching stage of the NCBI BLASTN software program. For example, for the 100kbase query sequence, the hits from our FPGA design can generate 1270 HSPs, while the hits from NCBI

Stage1 can only generate 1212 HSPs. In addition, we have further evaluated the performance on the gapped extension stage. Almost all HSPs generated by the FPGA pass the gapped extension stage, while, for NCBI BLASTN, a small portion of the HSPs are failed at this stage. Table 4.3 shows that our word matching accelerator design achieves a good sensitivity on hit generation. Beside sensitivity, we are also interested in the runtime performance of our FPGA accelerator, which is also the key concern of our design. Table 4.4 gives a runtime performance comparison between NCBI BLASTN and the DRC accelerator.

Table 4.4 Runtime performance comparison against NCBI BLASTN Stage1

Query	**10kbase**	**50kbase**	**100kbase**
DRC word matching* (Mbase/s)	1940	1580	756
NCBI Stage1 1 thread* (Mbase/s)	233	107	70
Speedup	8.3	14.7	10.8
NCBI Stage1 2 threads (Mbase/s)	466	200	140
Speedup	4.16	7.9	5.4
NCBI Stage1 4 threads (Mbase/s)	466	200	140
Speedup	4.16	7.9	5.4

*The throughput reported above does not include loading the database from the hard disk to the local RAM and other preprocessing stages.

Table 4.4 shows a significant speedup compared to the single threaded NCBI software. We have also tested the throughput of the NCBI BLASTN Stage1 using different number of threads. As the computer used in this paper does not support hyper-threading, the two cores can process at most two threads at the same time. This explains why there is no performance difference between 2 threads and 4 threads for NCBI BLASTN. The maximum query sequence size that our design supports in a single database scan is 100kbase. For query sequences longer than 100kbase, we have to split the query sequence into multiple segments (e.g., t segments, where each segment is $\leq$ 100kbase). Then, the database needs to be scanned t times to complete the search process (Mercury has the same problem and

also splits large query sequences into segments of 10k bases due to limited on-chip BRAM resources). This naturally leads to a *t*-time slowdown in the effective throughput. If more on-chip memory space was available, the Bloom filter could support longer query sequences in a single database scan.

When analyzing the speedups using different query sequences, one unusual situation is also observed. Table 4.4 shows a significant performance drop in speedup for the 100kbase query, which has a large number of hits. A possible explanation for this observation is that the Bloom filters need to stall the pipeline when the match hit FIFO is full if the Bloom filters generate too many hits (i.e. over one hit per clock cycle on average). This is further complicated by the limited bandwidth (maximum 100MHz × 64bit) to the off-chip memory in our Virtex-5 based experimental platform. Theoretically, the off-chip memory can only support one hash lookup table access each cycle at 100MHz. Thus, when there is a high hit rate, the off-chip memory access becomes a performance bottleneck. To show that our architecture is scalable in terms of the query sequence length, we have conducted another experiment (shown in Table 4.5) using a set of query sequences which generate fewer hits. The query sequences are randomly chosen from the *E.Coli* genome (NC_008253, 10kbase, 50kbase, and 100kbase, respectively).

Table 4.5 Performance comparison using *E. Coli* query sequences

Query	**10kbase**	**50kbase**	**100kbase**
# of hits (FPGA)	3,374,936	17,633,252	33,991,990
# of hits (NCBI Stage 1)	2,736,960	15,390,412	29,675,898
DRC word matching (Mbase/s)	2000	1780	1520
NCBI Stage1 1 thread (Mbase/s)	350	140	107
Speedup	5.7	12.7	14.1
NCBI Stage1 2 threads (Mbase/s)	700	280	200
Speedup	2.8	6.3	7.6

The performance drop between the 50kbase query and the 100kbase query for our FPGA design (Table 4.5) is only 14.6%, compared to a corresponding performance drop of 52% in Table 4.4. To examine the effect of match rate (or hit rate: defined as the number of hits divided by the database size), we define the expected number of match *w*-mers as the product of the match rate and the degree of parallelism. Then, for the hg18 100kbase query, the match rate for *w*-mer is about 5.86%. Sixteen *w*-mers are examined in parallel each cycle. Thus, 0.937 *w*-mers are expected to pass the Bloom filter stage each cycle (omitting the delays introduced by PPBFs and the effect of the local stalling mechanism to simplify the analysis), which approaches the theoretical upper bound of the off-chip memory access capability. If the off-chip table has to stop the pipeline to prevent hit loss, it will lead to drastic performance degradation. In contrast, for the *E. Coli* 100kbase query, the match rate is only about 2.4% and 0.388 *w*-mers are expected to pass the Bloom filters each cycle (well within the off-chip hash table's processing capability). When the supported query size (for a single iteration) is small, the number of expected match hits is small and the negative influence of match rate is negligible. However, longer query sequences can introduce a new constraint on future Bloom filter designs, which is not addressed in previous designs from the literature. In fact, the balance between the degree of parallelism, the maximum query size each iteration, and the match rate should be taken into account carefully. One possible solution to moderate the off-chip memory access bottleneck is to use memory with a higher bandwidth, such as DDR3, or provide multiple off-chip memory interfaces.

Furthermore, we have also recorded the NCBI BLASTN Stage1 performance using query sequences from the human genome on a state-of-art Intel Core i7-870 CPU containing four processor cores (2.93GHz) and 8GB RAM. The results are shown in Table 4.6. It can be seen that a better performance is achieved as more CPU cores are available. However, our DRC accelerator still reports a slightly better performance compared to the software version running with eight threads. We believe that with the help of newer FPGA technologies, better performances can be expected with the same architecture. As far as we know, the latest Virtex-7 FPGA

chip can provide up to 85Mbit of block RAMs (in contrast, our Virtex-5 chip only provides 10Mbit of block RAMs). This indicates that we could support much longer query sequences or construct the hash lookup table using the on-chip memories to give faster access speed. Meanwhile, the Virtex-7 chips use a faster data transfer interface (e.g., DDR3), which also provides the potential to further increase our design's performance.

Table 4.6 Throughput of NCBI Stage1 using Core i7-870 CPU with four cores

Query	**10kbase**	**50kbase**	**100kbase**
1 thread (Mbase/s)	233	127	82
2 threads (Mbase/s)	466	233	155
4 threads (Mbase/s)	700	466	280
8 threads (Mbase/s)	1400	700	466

4.6.2 Performance comparison against Mercury BLASTN

Although Mercury BLASTN is not the first FPGA design that aims to accelerate the NCBI BLASTN alignment algorithm, it gives a thorough analysis on the performance bottleneck of the software program. Mercury BLASTN also shows a highly efficient and dedicated architecture to accelerate DNA sequence alignments, which is very helpful for designers to understand the influence of different stages (or sub-stages) on the overall performance. Our design is also following the sub-stage structure given in Mercury BLASTN. Thus, it will be interesting to compare the merits of the two FPGA designs with a similar sub-stage structure.

Table 4.7 shows the Stage1 throughput comparison between the Mercury system and our DRC accelerator under different query sizes (the Mercury throughput, as reported in [60], also uses the mm8 1.4Gbase mouse genome and the hg18 query sequences). For the short query size (e.g., 10kbase query), both designs achieve a similar throughput. As the query size increases, our design achieves a higher throughput rate than the Mercury design. For large query sequences (> 100kbase),

the slowdown due to the need to split the query sequence into multiple segments can be clearly seen. To further explore the influence of larger query sequences, we also conduct an extra experiment with a query sequence of 9Mbase from the human genome against the mouse genome. The effective throughput is 8.2Mbase/s (roughly 90 times slower than the 100kbase query throughput). This additional experiment also proves the correctness of the t-time slowdown prediction described in section 4.5.1.

Table 4.7 Throughput comparison between the Mercury and the DRC accelerator

Query	**10kbase**	**50kbase**	**100kbase**	**1Mbase**
Mercury (Mbase/s)	1400	700	350	35
DRC accelerator (Mbase/s)	1940	1580	756	80.6

In Mercury, there are four Bloom filters working in parallel. Each Bloom filter has 6 hash functions and a 32kbit memory for the bit-array. In total, it consumes 24 hash functions and 128kbit block RAMs. In order to make a more qualitative comparison between two architectures, we have estimated the performance of our structure using the same amount of resources mentioned in the Mercury implementation. Table 4.8 shows the performance comparison with a 10kbase query sequence in terms of FPP.

Table 4.8 Performance comparison between the Mercury design and our 8×2 PBF with a 10kbase query sequence

	Mercury	**DRC 8×2**
Total size of BRAM consumed	128kbit	128kbit
Total number of hash functions	24	32
w-mers preprocessing in parallel	16	16
FPP	0.35	0.045

Table 4.8 shows that our 8×2 partitioned Bloom filter achieves a better FPP. We can deduce that, with the same FPP requirement, our 8×2 partitioned Bloom filter can provide a higher degree of parallelism. Furthermore, our Bloom filter works well under low match rate conditions. For query/database pairs with a high match rate, we can apply a 4×4 Bloom filter architecture to reduce the ingest throughput while having no influence on the FPP. A minor negative effect of the partitioned Bloom filter design is that, for a matching *w*-mer, it takes more cycles to complete all hash queries. However, different configurations of Mercury's Bloom filter structure have a much wider influence on the whole design. As its Bloom filter uses a conventional structure, modifications on its structure will change the total number of hash functions, the FPP, the degree of parallelism, and the number of matches per cycle. If the structure does not fit the constraint on the hash table lookup stage, hits loss will also appear.

The number of PPBFs applied in our PBF design determines the degree of parallelism. As each PPBF can support two hash queries at the same clock cycle, the system's degree of parallelism is $2\times P$ (P is the number of PPBFs applied in the design), theoretically. However, it is hard to proclaim that an 8-PPBF design is two times faster than a 4-PPBF design. Eight PPBFs working in parallel also indicates more frequent local stalls (introduced by the false positives and the true matches), which can make the 8-PPBF design's performance drop faster. In addition, if the 8-PPBF design generates more than one match each cycle, the performance will drop further to guarantee no sensitivity loss.

The influence of the match rate r also varies depending on different Bloom filter architecture configurations. Assume that we do not count the stalls generated by each parallel partitioned Bloom filter. Table 4.9 shows the expected number of matches that can pass the partitioned Bloom filters for different configurations. For example, for a match rate of $r = 0.05$, the 16×1 architecture can support 32 *w*-mer tests in parallel (each PPBF supporting two *w*-mer tests) and generate 1.6 matches each cycle, which exceeds the processing capability of the hash table access stage.

However, the 8×2 architecture has only 0.8 matches. Thus, the 8×2 architecture is a better choice under such a match rate.

Table 4.9 Expected matches for different architecture configurations

	2×8 arch	**4×4 arch**	**8×2 arch**	**16×1 arch**
Expected matches per cycle	4r	8r	16r	32r

If we take stalls into account, the analysis becomes more complicated. To simplify our analysis, we make the following assumptions: the match rate is r for each PPBF, the degree of parallelism is P and the number of hash functions in each PPBF is q (e.g., in the 8×2 architecture, P = 8 and q = 2); the FPP of each hash query is p, only one w-mer is examined for each PPBF. Then, we can establish an upper bound on the speedup (compared to the conventional Bloom filter architecture that processes 1 w-mer per cycle) as

$$S = \frac{P}{1+r(P-1)+(1-r)(\sum_{i=1}^{P-1} p^{qi})} \tag{4.6}$$

where $r(P-1)$ represents the true matches from other PPBFs, $(1-r)((\sum_{i=1}^{P-1} p^{qi})$ represents false-positive matches from other PPBFs ahead. If a w-mer passes one PPBF, it will stall the next PPBF data loading, which reduces the process speed correspondingly. For a single PPBF, the w-mer processing time is proportion to: the number of w-mers belonging to this PPBF, true matches from other PPBFs, and false positive matches from other PPBFs. Assume there are only P w-mers need to process. Thus, w-mer belonging to each PPBF is 1. The total true matches from other PPBFs will be $r(P\text{-}1)$. The false positive match analysis is a bit more complicated than the true matches. Assume $PPBF_1$ will send false positive matches to $PPBF_2$. The number of false positives from $PPBF_1$ will be $(1-r)p^q$ (each PPBF has q hash functions, thus its overall FPP is p^q). The false positive matches from other PPBFs will be further reduced by multiply the factor p^q. The total number of false positive matches for a single PPBF will be $(1-\mathrm{r})(\sum_{\mathrm{i}=1}^{\mathrm{P}-1}\mathrm{p}^{\mathrm{qi}})$. Therefore, the effective number of w-mers that our Bloom filter

design can support at each time unit is $S = \frac{P}{1+r(P-1)+(1-r)(\sum_{i=1}^{P-1} p^{qi})}$. From Equation 4.6, we get a rough performance evaluation of different architectural configurations. Figure 4.12 gives the estimated speedups for different architecture configurations.

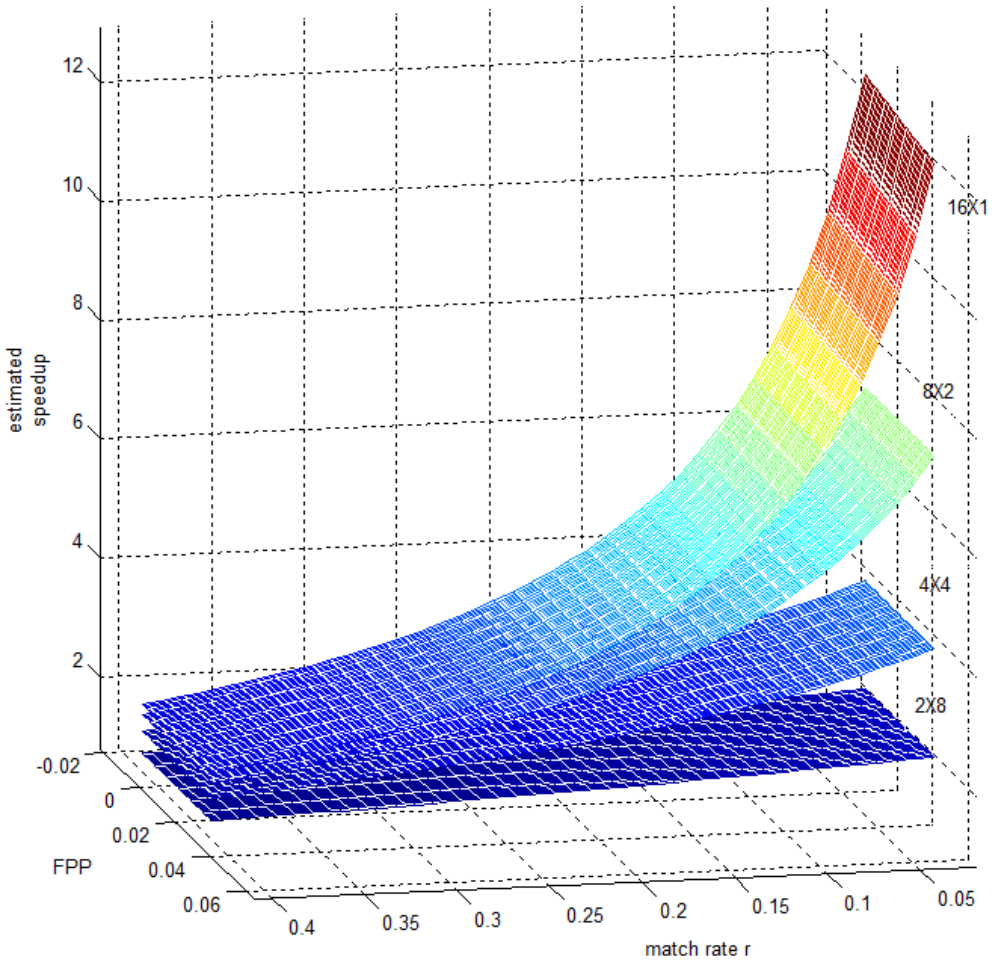


Figure 4.12 estimated speedups for different architecture configurations

It is clear that along with the decrease in match rate *r*, the estimated speedups improve dramatically for architectures with a higher degree of parallelism (the 16×1 architecture performance increases much faster than other architecture configurations). When the match rate is low, the speedups for different architectures are close to each other. However, many other parameters also influence the final speedup, such as the distribution of the matches and the processing capability in subsequent stages. For example, for r = 0.05 and p = 0.5,

the speedup S for the 8×2 architecture is 4.8 and S for the 16×1architecture is 5.97. Although $S_{16\times1}$ is slightly larger than $S_{8\times2}$, in practice, the 8×2 provides a better performance. If two configurations provide a similar S, it is safer, based on our experience, to choose the one with a lower degree of parallelism. Generally, the memory access is the word matching stage's performance bottleneck: the on-chip memory resources limit the number of hash functions working in parallel and the query sequence size supported, the off-chip memory limits the number of matches allowed each cycle.

The Mercury design uses a near-perfect hashing strategy to reduce the number of hash collisions (for 25kbase query, it consumes 16kbit on-chip memory and generates several hash collisions). However, based on our observations, when hash collision is reduced, duplicate *w*-mers become the dominating factor than hinders the performance of off-chip memory accesses. As the query size increases, the number of duplicate *w*-mers will be much larger than that of hash collisions. As duplicate matches cannot be eliminated by any hash function, we apply a simpler method, bucket hashing, to reduce the number of hash collision and mitigate the on-chip memory space consumption. Our test with a 25kbase query sequence (randomly chosen from the human genome) results in 222 hash collisions and 1975 duplicate *w*-mers using bucket hashing. In contrast, if we use a simple hash function with the same query dataset, it would produce 3461 collisions. Meanwhile, the on-chip memory consumption of our bucket hashing strategy is slightly bigger than 5kbit. Although bucket hashing produces more collisions than the near-perfect hashing, its influence on off-chip access is quite limited, when compared to that of duplicate *w*-mers. In addition, our on-chip quick lookup table consumes much smaller space than that of Mercury BLASTN (5kbit for our QLT, compared to 16kbit for Mercury's displacement matrix), which could avoid competing memory resources with the Bloom filter stage.

In general, our design can provide better computational efficiency under low match rate conditions than the conventional Bloom filter architecture. The performance

bottleneck lies in the memory access, which limits the throughput and parallelism for both the Mercury system and our word matching accelerator.

4.7 Summary

This chapter has presented an FPGA-based reconfigurable architecture to accelerate the word matching stage of NCBI BLASTN, which is a bio-sequence search tool of high importance to bioinformatics research. The performance comparison of our DRC accelerator to that of a single-threaded software implementation of NCBI BLASTN shows a speedup around one order of magnitude with modest resource utilization. The proposed system works especially well under low match rate conditions. The analysis of different architecture configurations reveals the performance bottleneck and the upper bound for speedups, which can be used as a guide for future designs.

Chapter 5

Introduction to Short Read Alignment

5.1 Background

Short read alignment and short read assembly are two important problems in the field of DNA sequencing, both of which refer to DNA reconstruction from DNA fragments. Short read assembly assembles the DNA fragments to create the original sequence (De novo assembly). Short read assembler design is usually based on the overlap graph method or De Bruijin graph method [97]. As there is no pre-knowledge on the reference genome, the assembly of short reads is a very difficult problem, especially when the volume of the DNA fragments becomes very large. Figure 5.1 gives an example on the short read assembly process.

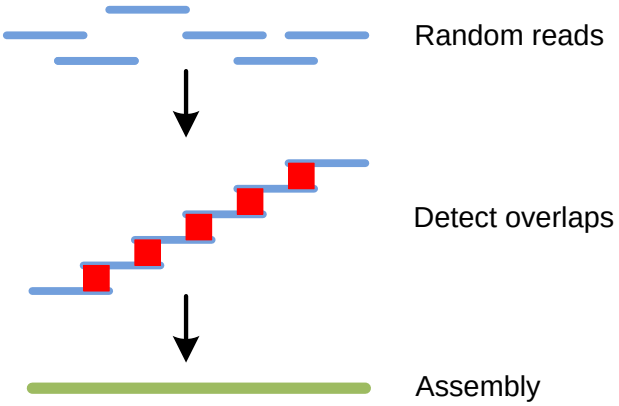


Figure 5.1 Short read assembly

Compared to the short read assembly, the short read alignment problem is much easier to implement. It is to find all locations in the reference genome that are identical or similar to the reads. Further analysis on these locations with mutations

can reveal the possible cause of diseases. Figure 5.2 shows an example of the short read alignment process.

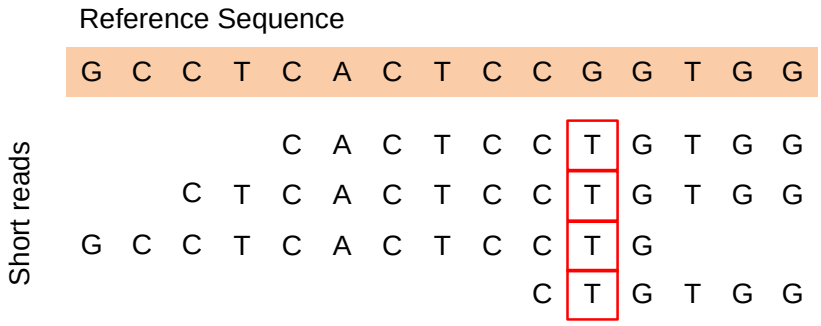


Figure 5.2 Short read alignment

In this thesis, we focus on solving the short read alignment problem. Short read alignment is similar to pair-wise sequence alignment. These short reads are generated through certain sequencing technologies. One of the most popular sequencing technologies is the shotgun sequencing technology [117, 10]. Shotgun sequencing works in two steps, read DNA fragments of size 500-700bp and assemble these short reads into a single long sequence. Although shotgun sequencing is one of the advanced techniques for genome sequencing from 1995 to 2005, the appearance of next-generation sequencing (NGS) technologies changes the situation. NGS technologies (such as Illumina sequencing [53] and SOLiD sequencing [86]) boost the short read production with dramatically lower unit cost (e.g. over one billion short reads (from 25-500bp) can be generated in one instrument run [127], but also introduce certain degree of sequencing errors at the same time). These sequencing techniques have led to an explosive growth in the size of short read datasets. Thus, balancing between the mapping quality and the execution speed becomes a new challenge. Restrictive error models are often used to improve computation speed, whereas more flexible error models are generally too slow for large-scale applications. As the read length continues to increase and more errors are permitted in the final alignment, many current aligners [67, 68, etc] are becoming less efficient.

5.2 Short read alignment solutions

Intrinsically, the short read alignment problem is still a string matching problem. Although dynamic programming based algorithms (e.g. the Smith-Waterman algorithm) can give an accurate answer to this kind of problem, it is too slow to apply directly for large volumes of data. The search space of a dynamic programming based algorithm is $O(mn)$, where m is the length of reference genome and n is the length of query sequence. As the reference genome size is large and thousands or millions of short reads are required to be aligned against the reference genome, how to effectively narrow down the search space becomes the dominant factor that determines the solution's performance. The general form of the short read alignment problem is given below.

Short read alignment problem:

Find out the locations with a high degree of similarity in the reference genome for each short read

Input: a set of short reads s_1, ..., s_n, a reference genome r, the number of maximum errors k

Output: all locations in the reference genome where each short read has at most k errors

---

Normally, three types of data structures are widely used to quickly narrow down the search space for the short read alignment problem. They are the hash table, suffix tree or suffix array, and BWT-based index.

5.2.1 Hash table

The string matching problem can be solved using a hash table data structure. The hash table is initialized with an empty table, working in two stages, the programming stage and the querying stage. In the programming stage, all substrings of length k from the reference genome must first be hashed to a certain location of the table, where k is a value smaller than the short read length. The

index of the substring and other related information should be stored in the hashed locations in the table. A hash collision might appear during the programming stage. A common solution is to apply the chaining technique, i.e. store the colliding elements using a linked list. Figure 5.3 gives an example of the chaining in a hash table. Suppose that ten integers (2, 5, 10, 12, 15, 20, 22, 32, 45, and 55) are required to be inserted into the hash table. The selected hash function generates only three hash addresses. Thus, if integers share the same hash address, they will be stored in a link list. The table structure can quickly determine whether the substring of the short read exists in the reference genome (constant access time, if the query is not hashed to a colliding location).

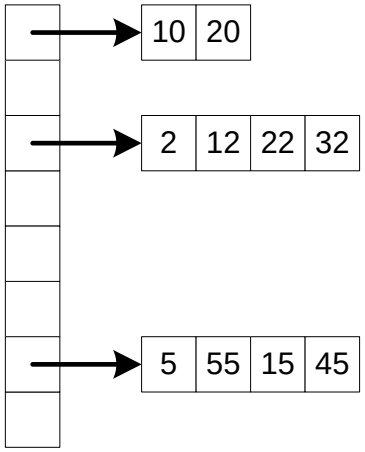


Figure 5.3 Chaining in a hash table

The hash table has a relatively simple structure and can provide significant speedup in practice. However, several drawbacks also limit its utilization. As the size of hash table is usually several times larger than the programming sequence, it will consume a huge memory space for long sequences. For example, a reference genome with one billion base pairs will require at least one billion entries to represent all its seed locations (omitting the appearance of hash collisions here to simplify the analysis). If each slot in the hash table is represented by a 32bit data word, it will require at least a 32Gbit memory for the complete table. For an even longer reference genome, the hash table might consume all memory on the host PC. In addition, as the memory access speed is fairly slow, the program's processing

speed can be limited by the intensive random accesses within the large hash table. When the hash collisions are taken into account, the accessing speed will be further reduced (this is because hash collisions require additional table access operations during the querying stage). Furthermore, although hash table is an efficient architecture to find out exact matches, it is quite difficult to solve the inexact matching problem use a single hash table, which will further increase the memory burden.

5.2.2 Suffix tree and suffix array

The suffix tree is another popular data structure suitable for the string matching problem. The suffix tree is a root tree structure constructed using all the suffixes of the text *T* of length *m*. Each edge represents a substring of the text. A preprocessing step, suffix tree construction, is required before the suffix tree search. The construction process will take $O(m)$ time using Ukkonen's algorithm[124]. To answer the question of whether a query string occurs in the text, it only takes $O(n)$ time, where *n* is the size of the query string. A typical example of string matching using a suffix tree is given in Figure 5.4 (a). The match locations of the substring "AT" can be quickly located with the help of the tree structure. Although the tree structure can be quickly constructed, it still occupies a lot of space. A further improvement of the suffix tree data structure is the suffix array (SA) which only stores the indices of the suffixes in a lexicographically sorted order. A special unique symbol '$', which is the smallest character among all characters in the original string *T*, is attached to the end of *T*. Figure 5.4 (b) gives an example of the suffix array data structure. The memory consumption of a SA construction ($O(m)$) is smaller than that of a suffix tree. The basic suffix array data structure takes $O(nlogm)$ time in the worst case to complete the substring search, where *n* is the size of the substring. However, enhanced suffix arrays (with the help of additional tables) can achieve an optimal time complexity of $O(n)$ [1]. The suffix tree and suffix array are highly efficient data structures for exact pattern search problems. However, when dealing with an approximate string matching problem, modifications on the search algorithm are required. One possible solution is to

search smaller exact match pieces first, which can lead to a tradeoff between the search speed and accuracy. An in-depth analysis on the suffix-based indexing method for approximate string matching is provided in [90].

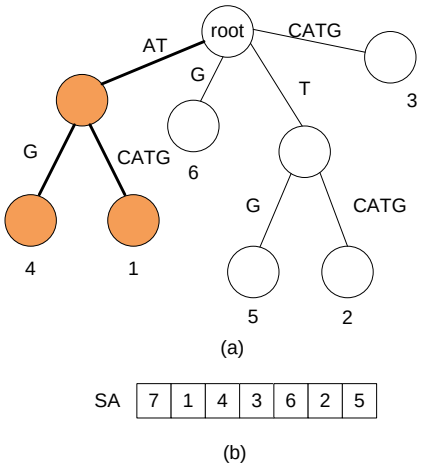


Figure 5.4 (a) Find string **AT** through the suffix tree for the text **ATCATG** (b) The suffix array of the text **ATCATG$**

5.2.3 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) [21] (or block-sorting compression) is an algorithm initially designed to improve the performance of data compression. When applying the BWT on a target character sequence, there is no influence on the sequence characters except that the order of them is permuted: substrings with a high occurrence rate will be grouped to the same region in a lexicographical order sharing the same initial character after transformation, which is very helpful to compress a sequence with many repeated characters. To give a better introduction to the Burrows-Wheeler transform, we take string "*ACAACG*" as an example (shown in Figure 5.5). Similar to the SA construction, a lexicographically smallest sentinel character "$" is appended to the end of the input text *T* to conduct the BWT computation. A Burrows-Wheeler matrix of *T* is generated, each row of which is a rotation of the input string *T$* and is sorted in lexicographical order. The substrings sharing the same prefix are permuted together (e.g. row 3 and row 4 in the Burrows-Wheeler matrix share the same prefix "*AC*"). The BWT matrix is only

an intermediate stage of the Burrows-Wheeler transform. In practice, the rightmost column of the matrix is stored as the transformed string of *T*, BWT(*T*). The index number indicates the corresponding character's position in the original text *T*.

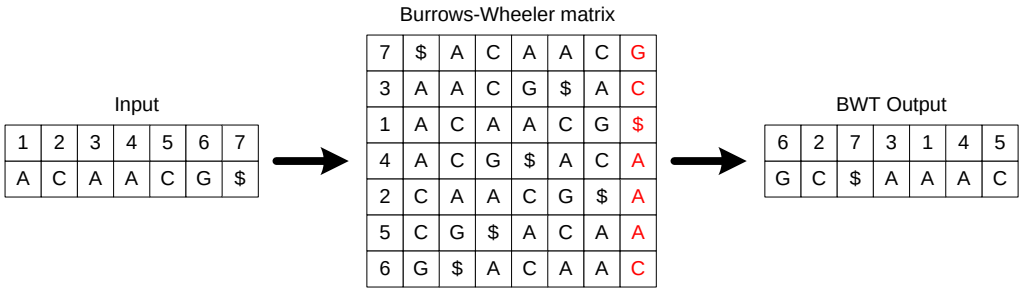


Figure 5.5 The Burrows-Wheeler transform of string "*ACAACG*"

One important property that BWT has is called "last first mapping": the i^{th} occurrence of a symbol *X* in last column of the BWT matrix is also the i^{th} occurrence of the same symbol in the first column of the matrix. The original string can be reconstructed using this property (see Figure 5.6). For example, the edge "G → C" indicates that character 'C' is before character 'G' in the original string. Then, we can get a substring "CG". Meanwhile, this character 'C' happens to be the second 'C' in the BWT output string, which leads to the next edge "C → A". For the same reason, we know that the character 'A' is before character 'C' and get substring "ACG". This search process terminates, when it gets the edge "A → $". Thus, the original string "ACAACG" is reconstructed.

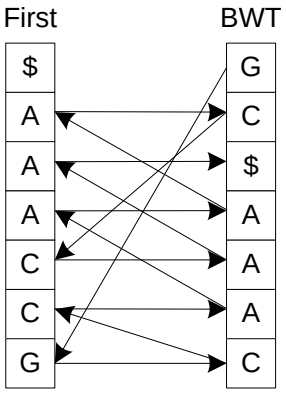


Figure 5.6 Text reconstruction routine

Based on this property, a backward search algorithm based on FM-index [41] is proposed. Two additional data structures are applied, a character table C and a occurrence table *Occ*(*c*,*q*). C[*x*] contains the total number of characters in text *T* that is alphabetically ≤ character *x*. *Occ*(*c*,*q*) is the number of occurrences of character *c* in prefix of *T'*[1..*q*], where *T'* is the output of *BWT*[*T*]. Define that **Fst** is the pointer to the start location of a substring in the matrix and **Lst** is the pointer to the end location of this substring in the matrix. To search a pattern *P*[1,*n*], the pseudo-code is listed as follows.

Algorithm backward_search(*P*[1,*n*]) [41]

i ← *n*, *c* ← *P*[*n*], **Fst** ← *C*[*c*] + 1, **Lst** ← *C*[*c*+1];
while (**Fst** ≤ **Lst**) and (*i* ≥ 2) do
    c ← P[*i*-1];
    Fst ← *C*[*c*] + *Occ*(*c*, **Fst**-1) + 1;
    Lst ← *C*[*c*] + *Occ*(*c*, **Lst**);
    i ← *i* - 1;
if (**Lst** < **Fst**) then return "no substring *P*[1,*n*] exists in *T*"
else return (**Fst**, **Lst**)

Taking the same sequence T="*ACAACG*" as an example, Figure 5.7 shows the search steps to find the pattern "*ACA*" following the backward search algorithm. The "F" and "L" symbol indicate the **Fst** and **Lst** pointer correspondingly (labeled in bold) in the Burrows-Wheeler matrix.

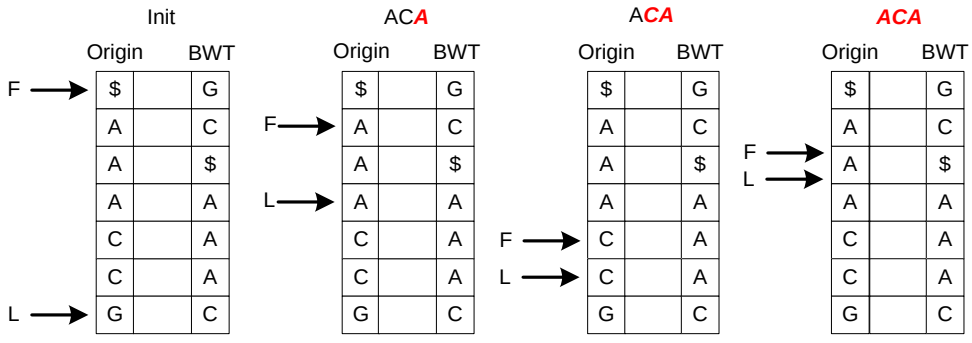


Figure 5.7 The backward search for "*ACA*" in the text "*ACAACG*"

As the search algorithm is proportional to the size of the query pattern, the time complexity is thus $O(n)$. The memory footprint is very small for a BWT-based backward search, which is also the key merit for its widespread utilization. Previous researchers have made a memory footprint comparison among different index methods targeting the human genome. The BWT-based indexation takes 1.3Gbyte to index the whole genome; a suffix tree requires over 35Gbyte; a suffix array requires over 12Gbyte; and a hash table consumes over 12Gbyte [61]. It is clear that BWT-based indexation achieves a significant memory saving over other indexation methods.

However, a common drawback exists for the data structures mentioned above. In molecular biology, mutation errors exist in the pattern (or short read), which leads to an approximate string matching problem. Three types of error are allowed, substitution, insertion and deletion (indel). The search space is thus increased according to the number of errors allowed. Under such circumstances, many strategies are proposed to accelerate the search process. Generally speaking, there is no perfect data structure for the short read alignment problem. Already, there have been many short read aligner designs based on the above data structures. A review on these designs can give us a better understanding of the design trends and can inspire new ideas for better solutions.

5.3 Previous work

"Seed and extend" is one of the most frequently used heuristics in short read mapping implementations. The basic idea is simple: since only a limited number of errors are allowed for a significant alignment, long exact match regions must exist. Discovering these exact matches (called common *k*-mers or seeds) can largely reduce the search space. The alignment errors can be further examined by applying the more computationally intensive Smith-Waterman-like alignment algorithms on the regions containing common *k*-mers. Detection of these common *k*-mers is usually performed using two approaches: (1) indexing of the input read dataset and scanning through the reference genome; (2) indexing of the reference genome and

aligning each read independently. Many tools have been developed utilizing these two approaches. However, in this section, we would like to categorize the alignment tools based on their implementation platform: software tools and heterogeneous tools.

5.3.1 Software alignment tools

BLAST [7] indexes the query sequence using a hash table data structure. The database is scanned with an 11-mer window (the default seed setting in BLASTN) to lookup exact matches, called seeds. The seeds are first extended without gaps to get high scoring pairs. Then, the SW algorithm is used to report local alignment. The BLAST program is a very large package, which can support different types of alignment. However, as it is not designed specifically for the short read alignment problem, its performance is quite slow compared to other popular alignment tools in this field.

Examples of short read alignment tools based on read dataset indexation are MAQ, ZOOM, and SHRiMP. Past researches [84, 72] have shown that a spaced seed model can provide extra sensitivity in genome search. In fact, it can also benefit the computation of short read alignment. MAQ [67] uses six hash tables to index the read datasets to ensure that a sequence with ≤ 2 errors could be found. Meanwhile, each hash table is built by a specific spaced seed model, which improves the search sensitivity. Only the first 28bp of the reads (typically, the short read segment with fewer sequencing errors) are indexed to speed up the alignment process. MAQ introduces the quality score to evaluate the mapping quality and it also supports pair-end alignment. The pair-end mapping information can further improve the aligner's sensitivity. However, MAQ is only targeted to the short read datasets of length 30-40bp, and its performance for longer short reads is unknown. As multiple hash tables are applied, increasing the size of the short read dataset might further increase the memory consumption.

ZOOM [77] minimizes the number of spaced seeds without any sensitivity loss for short reads of length 15bp to 64bp. Multiple spaced seed models are applied to

provide greater hit rate. For example, four spaced seeds with a weight of 13 can identify all two mismatch hits for 33bp reads. ZOOM constructs hash tables for the read set and scanned the reference genome. Different versions of the original ZOOM program extend its functionality: ZOOM-C combines a confidence score into mapping; insertion and deletion are allowed in ZOOM-I; ZOOM-P is designed for pair-end alignment.

Three techniques are applied in the design of SHRiMP [110]: spaced seeds, *q*-gram filters, and vectorized Smith-Waterman algorithm. The spaced seeds provide higher sensitivity than the consecutive seed model. The *q*-gram filter can quickly isolate reads with multiple seed matches with a small window of the genome (i.e. identifying regions with high similarity), which largely narrows the search space. The vectorized SW algorithm utilizes "vector" execution unites to accelerate the alignment computation. Another interesting aspect within SHRiMP is that it supports color-space alignment (ABI SOLiD).

Examples of tools based on reference genome indexation are SOAP, WHAM, BFAST, and GASSST. SOAP [73] is an early short read alignment tool targeting to reads of length 25-50bp. By default, at most two mismatches or one continuous gap with a size ranging of 1~3bp are accepted. A simple hash table is constructed based on the reference genome using the consecutive seed model.

WHAM [76] uses a very special strategy to index the reference genome. The concept of fragment concatenation is utilized to reduce the number of hash collisions and space cost for hash table construction. WHAM supports alignment with gaps by allowing the indexed fragments to slide several characters from its original position. As the sequence data stored in WHAM is represented in a compact binary format, the conventional Needleman-Wunsch alignment on the character domain coverts to a series of bitwise operations, which can be efficiently implemented by modern processors. WHAM achieves a faster processing speed than Bowtie (one leading state-of-art aligner widely used for short read alignment). However, its performance drops to slower than Bowtie, when the number of allowed errors increases.

A BLAT-like fast accurate search tool (called BFAST) [51] is designed to support short read data produced by various sequencing platforms, such as Illumina and ABI SOLiD. Multiple indices with a spaced seed model are applied to increase the sensitivity of alignment. For example, for the human genome and a read dataset of length 50bp, ten indices are used. One of thes key advantages of BFAST is that it achieves a greater sensitivity and accuracy while maintaining an adequate processing speed, compared to other short read alignment methods.

GASSST [108] follows the conventional "seed and extend" strategy to conduct the short read alignment. Different from other aligners, GASSST inserts an additional stage, the pre-filtration stage, between the seed stage and the conventional dynamic programming based extension stage. Different types of pre-filters are integrated into the pre-filtration stage, the tiled-NW (TNW) filter and the frequency vector (FD-vec) filter, to quickly eliminate irrelevant hits from the seed stage. Each layer of the pre-filters estimates the lower bound of the number of errors with increasing accuracy and complexity. The first layer filter is built within the L1 cache, which can provide the fastest access speed compared to the main memory. Compared to BFAST, GASSST has a simple data structure for reference genome indexation. Although GASSST has a slightly lower sensitivity than BFAST, it achieves a similar processing speed to that of BWA (one of the fastest short aligners to date). The strategy applied in GASSST gives a novel solution to the short read alignment problem.

The third approach, based on the Burrows-Wheeler transform, addresses this problem by applying an occurrence table and a suffix array to store the reference genome in a space-efficient manner. Bowtie [65] employs a Burrows-Wheeler index which greatly reduces the memory consumption. Bowtie is one of the fastest alignment tools for short read alignment, but does not allow for indel errors. As the original backward searching algorithm only permits exact pattern matching, Bowtie applies a greedy depth-first search algorithm to support substitution errors. Both the forward and the reverse sequence of the reference genome are indexed

using BWT and a "double-indexing" technique is applied to mitigate the excessive backtrackings in the greedy search.

BWT-SW [63] is the first software tool designed to accelerate the dynamic programming based local alignment algorithm with the help of BWT indexation. BWT indexing is applied to reduce the memory consumption of the suffix tree data structure while preserving a similar searching time. It uses the dynamic programming algorithm to compute the optimal alignment score between the query sequence and each substring within the suffix tree. A simple pruning strategy is applied to terminate dynamic programming as soon as the number of alignment errors exceeds the predetermined threshold. Experiments show that BWT-SW is much faster than the conventional dynamic programming algorithm. While BWT-SW is able to find all local alignments, it is still several times slower than BLAST for long patterns.

BWA [68] also uses the BWT-based backward search to find inexact matches between the reference genome and short reads. A special function is used to estimate the lower bound of the number of differences (substitutions or indels), which improves the efficiency of the search algorithm. Several aspects distinguish BWA from other BWT-based algorithms: (1) different penalty scores are used for substitutions, gap openings, and gap extensions; (2) BWA conducts a breadth-first search with a heap-like data structure; (3) an iterative strategy is applied to accelerate computing while maintaining the mapping quality; (4) limited errors are allowed in the seed of a read, which further improves the processing speed. Generally, BWA achieves a very fast processing speed with a high sensitivity.

BWA-SW [69] is a tool designed for long read (several hundred base pairs) alignment. Unlike short read alignment, long reads are more fragile to structural mutations and assembly errors. The long reads might contain more gaps, which intrinsically expands the search space. BWA-SW also follows the "seed and extend" strategy. In BWA-SW, both reference genome and query sequence are indexed using the FM-method (a prefix trie for the reference genome and a directed acyclic word graph (DAWG) for the query sequence). Low-scoring matches are

pruned at each node, which reduces the scale of dynamic programming dramatically. Another optimization made in BWA-SW is that only non-overlapping query sequence alignments are reported. BWA-SW achieves much faster processing speed than BLAT [57] and SSAHA2 [95] with higher or similar accuracy.

SOAP2 [74] uses a bidirectional BWT to build the index of the reference genome. For an exact match, a hash table is applied to accelerate the location search process in the BWT reference index. For inexact matches, a "split-read" technique is applied to identify the mutations in the short reads. For pair-end alignments, it first aligns the paired reads independently. Then, refine the hits based on orientation relationship and distance. SOAP2 achieves much faster processing speed than SOAP with much less memory consumption, but its processing speed is slightly slower than Bowtie.

5.3.2 Heterogeneous alignment tools

Generally, short read mapping, particularly for large datasets, is computationally challenging. As the short reads are independent to each other, it provides the opportunity for massive parallel mapping. Thus, heterogeneous systems become a competitive candidate for this kind of computation task. Already, we have seen the development of hybrid short read alignment tools using the parallel computing capabilities of GPU to improve the algorithm's runtime performance.

SARUMAN [16] is designed to first identify all match positions for each read under a given error threshold, and then, to generate one optimal alignment for each hit position without any heuristic. It works in two phases: the mapping phase and the alignment phase. The mapping phase applies a q-gram filter to find candidate alignment positions on the host computer. The alignment phase executes the modified Needleman-Wunsch algorithm on the attached graphics card. As no heuristic is added in the alignment process, SARUMAN guarantees full sensitivity while maintaining a high processing speed.

CUSHAW [81] is a CUDA compatible parallelized short read aligner utilizing the BWT and FM-indexation to improve its runtime performance and alignment sensitivity. CUSHAW applies a depth-first search strategy to reduce the memory consumption of each thread on the GPUs. During alignment computation, different types of memory are delicately organized to maximize the performance. Two types of alignments are supported: seeded alignment and end-to-end alignment. Alignments with gaps are not allowed in CUSHAW. Compared to other popular aligners, CUSHAW achieves a better alignment quality for both single-end and pair-end alignments with a faster processing speed.

SOAP3 [78] is the newest version of the SOAP series, which combined the advantages of BWT-based indexation and parallelism provided by GPUs. As it is not a heuristic-based solution, no sensitivity loss will appear. SOAP3's optimizations are focused on two aspects: (1) reducing the randomized heavy memory access required by the original BWT search, while maintaining the search efficiency; (2) fully utilizing the GPU's processors by processing hard patterns (i.e. patterns with too many search branches) separately. SOAP3 achieves 7.5 to 20 times speedup over BWA and Bowtie, respectively. However, SOAP3 currently only supports up to 4 mismatches, which may limit its application.

Just like GPUs, FPGAs are also competitive candidates for massive parallel data processing. Many designs have proven that the runtime performance of the Smith-Waterman algorithm can be largely improved with the help of FPGAs. The data dependency for the conventional SW algorithm is not very strong. Cells located at the same anti-diagonal can be computed simultaneously (the black line shown in Figure 5.8), which is the basis for the parallel implementation of SW algorithm.

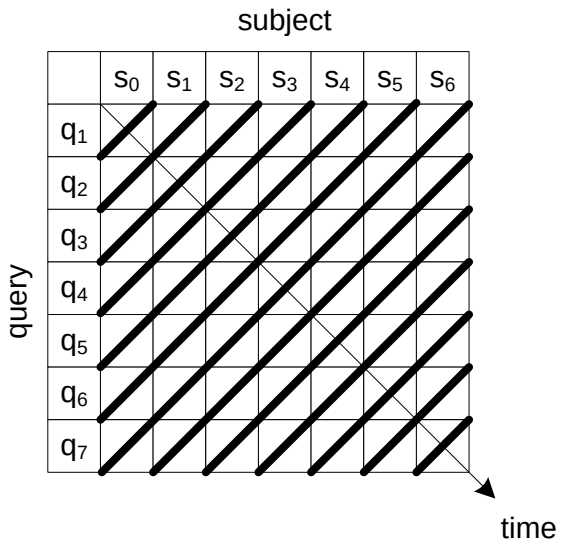


Figure 5.8 Data dependency in the alignment matrix

As the parallel processing cells are distributed in a regular format in the alignment matrix, the systolic array is a good choice for high speed implementation on an FPGA. The processing elements (PEs) were allocated horizontally in [134] (responsible for the column in Figure 5.8). Usually each PE will occupy 4 slices for implementation. One solution to reduce the resource consumption is to utilize runtime reconfiguration. However, this is a slow process and results in the performance degradation. Instead of runtime reconfiguration, the distributed RAM space can be utilized resulting in a PE design which consumes only 3 slices without any performance compensation. Another advantage is that the PE design is quite standard and can be used for ASIC designs. In the final system, 4,032 PEs are placed on an XCV1000E-6 device with an estimated highest clock frequency of 202MHz.

The conventional Smith-Waterman algorithm is implemented in [71] on an Altera FPGA with a Nios II soft processor to control the computation flow. Four working steps are included: (1) load the subject sequence and query sequence to local memory; (2) initialize the alignment matrix; (3) encode both the subject sequence and query sequence and load them to the 64×SCM block (the 64×SCM block consists 64 single cell modules (SCMs) supporting an 8×8 SW alignment); (4) perform the alignment computation. For longer sequences, the alignment matrix

can be partitioned into multiple 8×8 blocks and with the computation performed repetitively. The final system achieves up to a 160× speedup compared to the same algorithm running on a general purpose computer.

Although a lot of FPGA-based architectures for the SW algorithm have been proposed to improve the alignment computation's runtime performance, systems specially designed for short read alignment are rare.

A naive string matching approach is applied in [58] to find out the similarity between short reads and the reference genome. The reference genome segment is compared to multiple short reads stored on the chip. Low-level distributed RAM is utilized to simplify the base pair comparison process. A Gigabit Ethernet interface is applied to transfer data between the host PC and the FPGA board. The Virtex-6 FPGA chip supports up to 600 short reads of length 64bp for a single reference genome scan working at a frequency under 200MHz. Compared to Bowtie (both of them do not support gaps), the FPGA design achieves a similar processing speed but with 100% sensitivity. However, gaps are not allowed in this design.

An FPGA-based system is designed in [4] to align the query sequence to an ancestral state vector. The kernel algorithm is a parsimony-based phylogeny-aware short read alignment (PaPaRa) [14], which is similar to a dynamic programming algorithm but designed for evolutionary tree analysis. It uses the gigabit ethernet interface to provide fast data transfer between the host PC and the Virtex-6 FPGA. Score processing units (SPUs) are implemented to accelerate the alignment kernel and the trace-back steps are computed using the host PC. The final system achieves over 70x speedup compared to the original algorithm.

Fernandez *et al.* [40] designs a short read aligner based on BWT-based indexing. Both of the I-table (storing the first occurrence of each character on the sorted BWT(*ref*)) and the C-table (storing the count of each character on a previous location of BWT(*ref*)) are stored using the on-chip memory resources. An additional sampled suffix array is applied to indicate the start location of the pattern in the reference genome. A finite state machine is designed to control the

data transfer from different storage structures. This design achieves two orders of magnitude speedup compared to Bowtie. However, several drawbacks largely limit its application: first of all, the limited on-chip memory resources restrict the supported reference genome length to several hundred thousand bases (490,000 bases reported in this design) which is too small for real reference genomes. Second of all, mismatches are not supported in this design, which makes it a bit difficult to compete with current mainstream software-based short read aligners.

A heterogeneous architecture is presented in [121], which is based on the algorithm described by PerM [24], to accelerate the computation of short read mapping. In this design, the host CPU discovers reads with high similarity to the reference using a hash index. Then, the candidate reads are delivered to the four-stage processing elements (PEs) in parallel via a broadcast bus. The stored results are reads with fewer alignment errors than the predetermined threshold. As 100 PEs were applied, the design can thus support a maximum 100 (*read, ref*) pairs alignment at the same time. This design achieves an order of magnitude speedup compared to the PerM software. However, gaps are not allowed in this design. As the Smith-Waterman algorithm increases the search space and resource consumption dramatically, it might largely limit the number of PEs supported on the same chip.

An FPGA short read aligner is shown in [99] based on the algorithm used in BFAST. Two tables, a pointer table and a candidate alignment location (CAL) table, are constructed to index the reference genome. For each seed, both the forward and the reverse complement string are generated, but the implementation only uses the lexicographically smaller one to conduct the table lookup. A CAL filter is designed to avoid repetitive evaluation on the same (*read, ref*) pair. The Smith-Waterman algorithm is implemented using a systolic array structure to reduce computation complexity. The final system consists of eight Virtex-6 FPGA chips and 32GB DDR3 off-chip memory. Each of the FPGA chips is connected via a PCIe x16 network. This aligner achieves two orders of magnitude speedup compared to the original BFAST algorithm and an order of magnitude speedup

compared to Bowtie. One minor drawback for this aligner design is on its Smith-Waterman algorithm implementation: 18 alignment engines are required to handle the rate of generated candidate pairs. A more efficient architecture for the Smith-Waterman algorithm implementation can largely reduce its resource consumption.

5.4 Short read mapping analysis

The short read alignment problem is defined by two major aspects: (1) the large volume of datasets; and (2) the complicated alignment algorithms need to support different types of mutation errors. The naive approach for short read alignment is quite inefficient: all segments from the reference genome are considered to have equal importance. However, in practice, a short read will only achieve a high alignment scores at a small number of locations in the reference genome. Thus, most of the computational effort is wasted. Heuristics can largely improve the computation efficiency with a certain sensitivity loss. A well designed heuristic can boost the runtime performance while keeping the sensitivity loss within the acceptable scope. The search space for BWT-based heuristics will increase dramatically if more errors are allowed. Meanwhile, the computation for the backward search requires intensive table accesses (shown in section 5.2.3) and the table size can be larger than the on-chip memory size of an FPGA depending on the reference genome used, which is unlikely to be suitable for direct FPGA implementation. The hash-based heuristics separate the search process into two stages: the seed generation stage and the extension stage. The "seed" stage is to find the candidate regions with a high degree of similarity between the reference genome and the query read sequence. The conventional method is based on the hash strategy. It is simple and accurate, but it can generate too many candidate locations, which will put a significant burden on the subsequent extension stage. The extension stage extends the seed in both directions utilizing the DP-based NW algorithm. Before introducing the search algorithm of our FPGA implementation, we conduct a simple test to find the performance bottleneck in the conventional NW-based short read mapping. We record the runtime of three different stages, i.e. indexing, seed generation, and seed extension in the NW algorithm on a

conventional CPU using a single thread. In this test, we use one million simulated reads of length 76 each from the *E. Coli* genome and a 4% error rate. The results in Table 5.1 show that the extension stage is the most time-consuming operation. As the extension stage has a regular systolic architecture, this makes it suitable for FPGA implementation. However, the influence of the seed generation stage on the overall runtime performance is still vague, as in Table 5.1, it only occupies 13.5% of overall runtime and it consists of random memory accesses to a large lookup table.

Table 5.1 Runtime performance for short read mapping with single thread on a quad-core AMD 2.1GHz CPU

	Runtime (sec)	**Percentage of time spent**
Indexing	7.03	0.79%
Seed generation	120.14	13.5%
Extension	760.19	85.7%

One method to moderate the extension stage computation is to add pre-filters designed to rule out areas with too many errors. The pre-filters are usually easy to compute but less stringent than the NW search. In our design, we choose a slightly different method to reduce the search space. In practice, a good short read alignment only allows a few errors (normally, the number of indels allowed is much less than that of substitutions). This provides us the opportunity to use another DP-based algorithm with a much smaller search space, the banded NW algorithm.

For two given bases x and y over the nucleotide alphabet $\sum = \{A, C, G, T\}$, we use the following scoring scheme: 0 for a match (i.e. if $x = y$), and 1 for a mismatch (i.e. if $x \neq y$). Furthermore, a value of 1 is charged for each indel error. The computation of the DP alignment matrix between a read sequence ($read[1..n] \in \sum^n$) and a substring of the reference genome ($ref[1..m] \in \sum^m$) is given in Equation (5.1) for $1 \leq i \leq m$ and $1 \leq j \leq n$.

$$S(i,j) = min\begin{cases} S(i-1,j)+1 \\ S(i-1,j-1)+\delta \\ S(i,j-1)+1 \end{cases} \quad (5.1)$$

where $\delta = 0$ if $ref(i) = read(j)$; otherwise, $\delta = 1$. The DP matrix initializations are given by $S(i,0) = i$ and $S(0,j) = j$ for $0 \le i \le m$ and $0 \le j \le n$. The optimal global alignment score with respect to the given scoring scheme is then the value $S(m,n)$. In fact, short read alignment can be considered as a semi-global alignment, where only the query read needs to be globally aligned. This alignment can also be computed using Equation (5.1), but the optimal score is given by the minimal value in the last column of S, i.e. $\min\{S(i,n) \mid i \in \{1,\ldots,m\}\}$ and the gaps in the first column are thus omitted. The search space for conventional global alignment is of size $m \times n$. The banded global alignment can limit the search process around the main diagonal. Figure 5.9 shows a typical case for banded semi-global alignment with a band width of one. The alignment score indicates the path with the least errors.

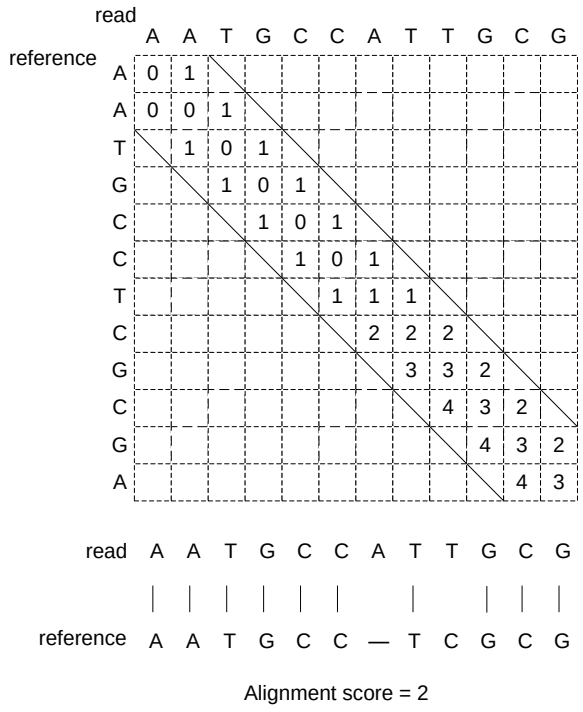


Figure 5.9 The conventional banded semi-global alignment

5.5 Summary

As an important topic in next generation sequencing, short read alignment has a wide influence on the development of multiple areas in genomic analysis. Different data structures have been applied to improve the search efficiency of the short read alignment problem, and we have also seen various alignment tools developed either on general-purpose computers or on heterogeneous platforms. A review on the past designs provides us a better definition on the short read alignment problem, which can be a useful guide for a new aligner design which we will present in the next chapter.

Chapter 6

Short Read Aligner Design and Implementation

6.1 Introduction

As mentioned in Chapter 5, the short read mapping problem is suitable for parallel implementation, as there is no data dependency among different reads. In this chapter, we concentrate on accelerating the short read alignment computation utilizing the parallel computing capability of FPGAs. Following the conventional "seed and extend" strategy, the runtime profile shown in Table 5.1 indicates that DP-based semi-global alignment is the most time-consuming stage. To better utilize the fine-grained pipelining and massive parallelism within FPGAs, we propose a block-wise alignment algorithm (implemented in the *Align Core* module) to approximate the score of the conventional dynamic programming algorithm. To evaluate the performance improvement of *Align Core,* we compare it against the extension stage of another popular tool GASSST, as both are built using the same "seed and extend" strategy. As the computation time for the extension stage is largely improved, we further integrate *Align Core* into a hybrid system and analyze the performance bottleneck for different system design strategies.

6.2 Short read mapping on an FPGA

6.2.1 Align Core design

The banded global alignment algorithm largely reduces the search space, but the alignment score is still computed sequentially. In order to further utilize the parallelism inherent in FPGAs, we have modified the banded semi-global alignment to a fine-grained parallel version. Instead of computing the alignment

score directly using Equation (5.1), we first take only substitution errors between bases into account. The highlighted bases in Figure 6.1 indicate the best alignment (the best alignment is also highlighted with a solid line) between two sequences. As a good alignment permits only a limited number of errors, the optimal alignment score is usually related to the path with fewer 1's. This gives an insight as to how to conduct the alignment in parallel. As the paired match information is independent of each other, we can divide them into different blocks and compute the alignment score for each block in parallel. In each block, the score computation is still following Equation (5.1). As it is difficult to guarantee that the best alignment in the block is always part of the final alignment, we compute the alignment scores for three different diagonals at the same time within each block, labeled as (S_u, S_m, S_l) to prevent possible sensitivity loss. Meanwhile, the start location (P_u, P_m, P_l) for each alignment in the block is also recorded. The start location will be used as a guide to concatenate two consecutive blocks. Assume that the band width is set to 1; the score for block1 is (S_{1u}, S_{1m}, S_{1l}) and the corresponding start location is (P_{1u}, P_{1m}, P_{1l}); the score for block2 is (S_{2u}, S_{2m}, S_{2l}) and the start location is (P_{2u}, P_{2m}, P_{2l}). The rule used for concatenation is as follows:

---

for each score in block2, $i = \{u. m, l\}$
    if its start location is u
        $S = S_{2i} + S_{1u}$
        if (S > threshold) eliminate the alignment
        else save the score and update the start location with P_{1u}
    if its start location is m
        $S = S_{2i} + S_{1m}$
        if (S > threshold) eliminate the alignment
        else save the score and update the start location with P_{1m}
    if its start location is l
        $S = S_{2i} + S_{1l}$
        if (S > threshold) eliminate the alignment
        else save the score and update the start location with P_{1l}

---

where the threshold is the maximum number of errors allowed for an alignment. Since the block scores are independent of each other, we can use a tree structure to concatenate multiple block scores in parallel to reduce the computation time. With the help of the block concatenate rule, the parallel block alignment algorithm

approaches a similar degree of similarity to that of the conventional banded NW algorithm. Our parallel block alignment algorithm is similar to the Four-Russian speedup technique [11] for block alignment, but we employ a different block partitioning strategy. Instead of partitioning the search space into overlapping square blocks, we divide the search space into consecutive "v-shaped" regions, without any overlapping. GASSST also introduces a tiled-NW filter to compute the alignment score in blocks. A small lookup table for 4bp long sequence pairs is applied for the block score computation. The lower bound score is also computed among three overlapping regions along the main diagonal. As this tiled-NW filter is the first stage for the cascading filters in GASSST, its performance for candidate hit elimination is quite limited. In contrast, our parallel block alignment algorithm can provide a high sensitivity on candidate elimination (see the experiment results). Figure 6.1 gives an example of the parallel block alignment computation for two 12bp sequences. The bold lines in Figure 6.1 construct three blocks for alignment computation. After block reformation, we get a substitution matrix for each block. The alignment score computation in each block is determined by its substitution matrix. The start locations are labeled as $\{u, m, l\}$ representing the upper diagonal, the main diagonal, and the lower diagonal, respectively. The threshold score is set to three. If the alignment score is higher than the threshold, the alignment will be ignored and labeled with the symbol "×". The block score computation is divided into two categories: the initial block alignment and the subsequent block alignment.

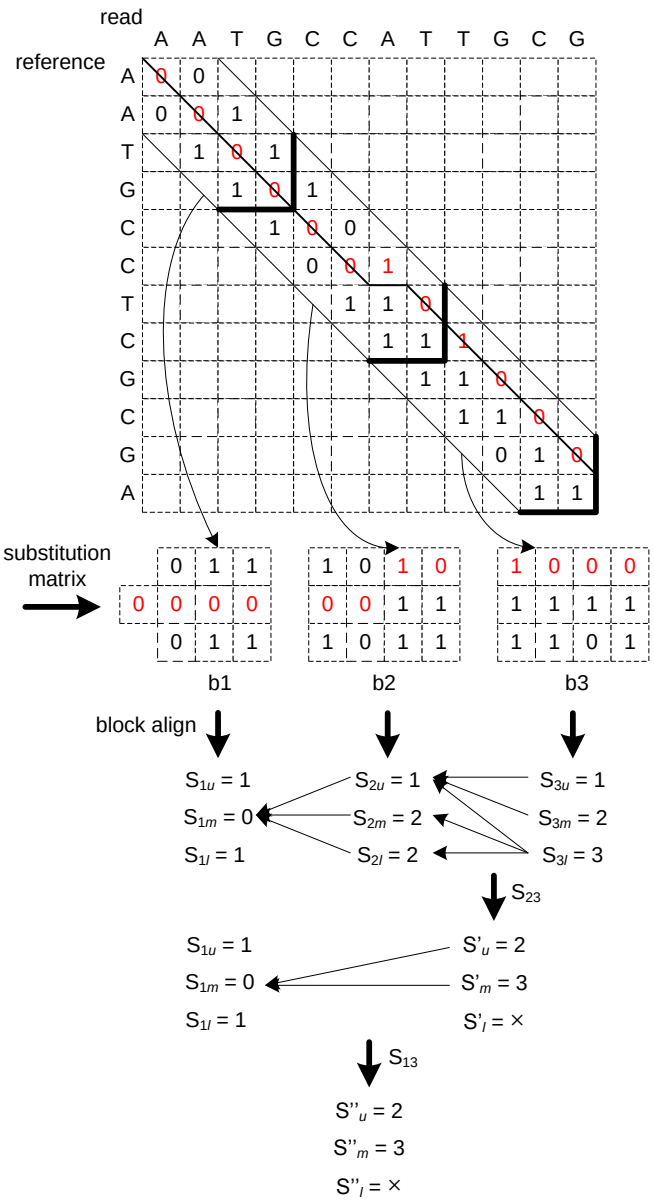


Figure 6.1 The parallel banded semi-global alignment

The initial block score computation is the same as the conventional banded semi-global alignment. The block b_1 score computation in Figure 6.1 falls into this category. Its alignment score is S_1 = (1, 0, 1) and the start location is (*m*, *m*, *m*). The score computation for block b_2 and block b_3 falls into the other category (the subsequent block alignment), and also follows Equation (5.1), but assumes that there are three possible start points (*u*, *m*, and *l*, respectively) for the alignment. Figure 6.2 shows the detailed computation steps for block b_2. The dashed arrows in

Figure 6.2 indicate the trace-back paths of the alignment. Based on these paths, we can determine the start location for each score in the last column of the score section. The final score for block b_2 is S_2 = (1, 2, 2) and the start location is (*m*, *m*, *m*). Following the same method, the alignment score for block b_3 is S_3 = (1, 2, 3) and the start location is (*u*, *u*, *uml*) (for the same alignment score, there might be multiple start locations). In Figure 6.1, we use the arrows to represent the start locations after block alignment. For example, the arrow from S_{3l} to S_{2u} indicates the start location of '*u*'; the arrow from S_{3l} to S_{2l} indicates the start location of '*l*'. Then, by concatenating block b_2 and b_3, we get the alignment score S_{23} = (2, 3, ×) and the new start location (*m*, *m*, *m*). Following the same strategy, we can get the score for all three blocks as: S_{123} = (2, 3, ×). Then, the final alignment score is 2 (the same as the conventional algorithm). For the conventional alignment algorithm, it requires at least 12 cycles to get the alignment score. In contrast, our parallel algorithm only requires 6 cycles to get the final score (4 cycles for block alignment and 2 cycles for score concatenation). Thus, as the read length increases, we could expect greater computational efficiency. In practice, we have designed an FPGA aligner including twelve blocks, where each block can conduct an 8bp alignment. The 8bp aligner can produce one result each clock cycle due to its fine-grained pipeline structure, with a corresponding pipeline latency of 8 cycles.

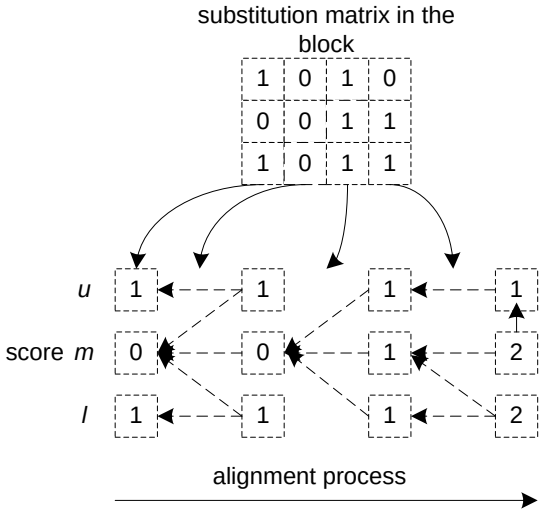


Figure 6.2 Alignment computation within the block

The *Align Core* has two working states: the initialization state and the processing state. In the initialization state, the *Align Core* is configured using setup commands. For example, the threshold score is set based on the *thresh_setup* command before alignment start. In the processing state, the (*read*, *ref*) pairs are input from the seed generation stage. The inner structure of *Align Core* is shown in Figure 6.3.

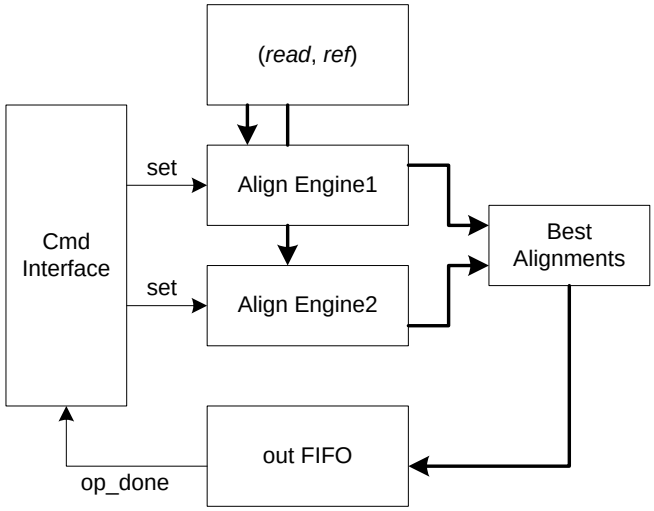


Figure 6.3 The *Align Core* inner structure

The *Command Interface* connects the FPGA aligner and the host PC through the HyperTransport interface. Customized commands (e.g. software reset, start, threshold setting, etc) are designed to control and monitor the FPGA alignment core's working state. The *align engine* implements the algorithm mentioned above to accelerate the alignment computation. Normally, a single alignment engine is enough to conduct the banded NW alignment, but in some occasional situations (characters outside the reference candidate region), the alignment score will be incorrect. Figure 6.4 gives an example of such special cases.

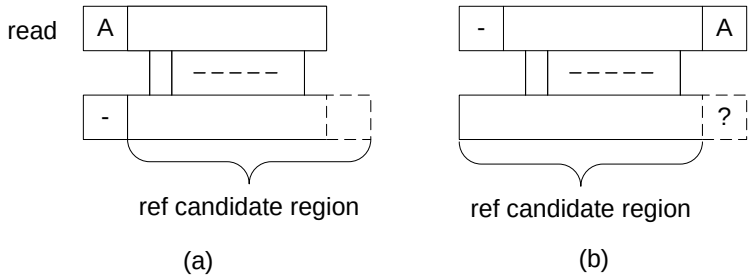


Figure 6.4 Special cases for semi-global alignment

Case (a) is missing the extra character to the left of the candidate region. In Case (a), the final score will be *S*+1 (*S* indicates the candidate region alignment score, 1 is the initial insertion) for the normal search. However, if the character to the left candidate region happens to be 'A', the correct alignment score will change to S. This will introduce a false negative answer in the final results. To prevent possible sensitivity loss, we expand the search space by applying two alignment engines. One is for the normal search; the other searches the reference region shifted by one base to the left. The "best alignments" module chooses the best alignment score between these two alignment engines and at the same time removes the duplicate results. The match information is then stored in the out FIFO. In contrast, Case (b) is missing extra characters to the right of the candidate region. As fewer nucleotides are computed, the alignment score could be less than its actual value, which adds false positive answers to the final results. To solve this problem, we also load one extra base to the right of the candidate region and compare it with the last character of the read. The alignment score from the lower diagonal will be updated with the extra base compare result.

The alignment engine design (shown in Figure 6.5) follows the parallel banded NW algorithm mentioned above. It consists of three components: a head processing mode, twelve smaller alignment modules (*Align8bp*), and a concatenate module.

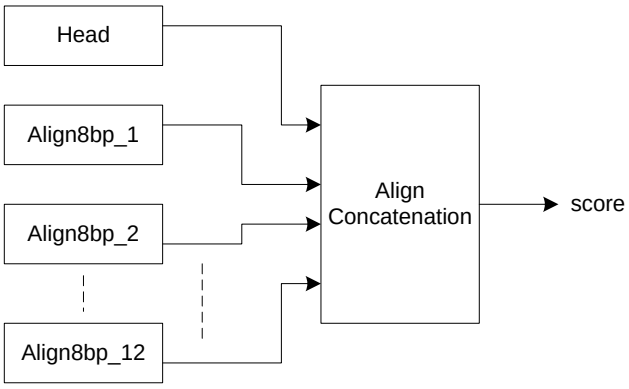


Figure 6.5 The alignment engine architecture

The head-processing module conducts the semi-global alignment for the first four bases. Twelve *Align8bp* modules can support up to a 96bp alignment computation. The *Align8bp* module conducts the alignment computation using a substitution matrix of size 3×8 for each block. Its inner structure is shown in Figure 6.6. The score of three diagonals are updated simultaneously. Each cell in Figure 6.6 is initialized with the value in the substitution matrix.

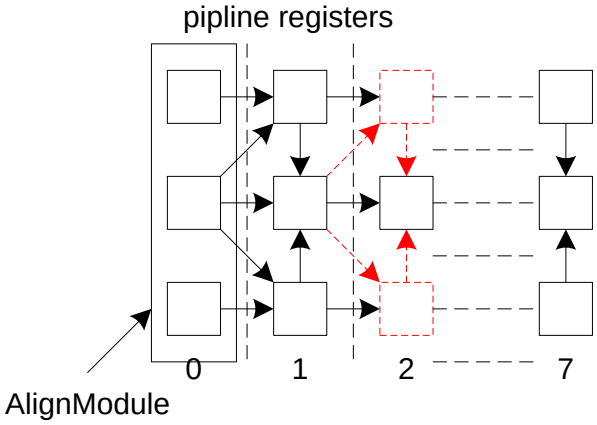


Figure 6.6 The *Align8bp* module inner structure

The alignment score is updated along the path labeled in the inner structure. Meanwhile, the original start point for the alignment score is also updated. In each cell, both the score computation and start location updating are related to the comparison between different input scores. When all computations are done, the last column will report the block alignment scores and corresponding start

locations. The most time consuming path for the *Align8bp* module is the main diagonal score updating and start location updating. We have labeled the critical path with dashed arrows in Figure 6.6. As the start location updating depends on the score computation, we implement an *AlignModule* to update the alignment information for three diagonals at the same time. An example is shown in Figure 6.7 to illustrate how we update the score and location information.

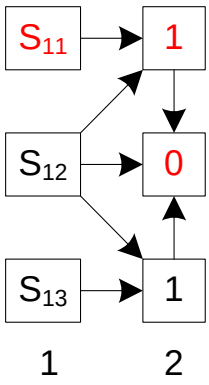


Figure 6.7 score and location updating

Assume that we already know the scores from the previous layer (S_{11}, S_{12}, S_{13}). To compute the scores for the next layer (S_{21}, S_{22}, S_{23}), we need to first compute S_{21} and S_{23}; and then S_{22}. Notice that $S_{21} = min\begin{cases} S_{11} + 1 \\ S_{12} + 1 \end{cases}$ and $S_{23} = min\begin{cases} S_{13} + 1 \\ S_{12} + 1 \end{cases}$. Thus, S_{22} can also be computed utilizing (S_{11}, S_{12}, S_{13}) and the match information (1, 0, 1). Based on this observation, we have designed our score updating structure (shown in Figure 6.8).

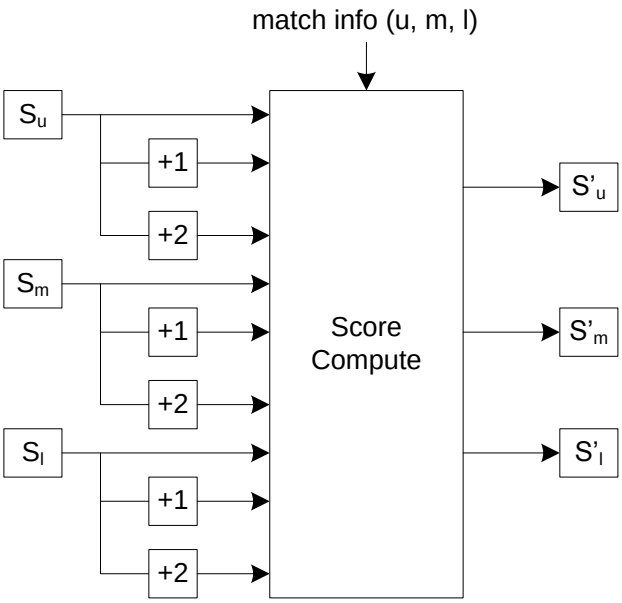


Figure 6.8 Score updating structure

The "+1" and "+2" modules are used to estimate different worst case conditions for each score (e.g. S_{22} can be generated from S_{11} with a mismatch and a gap). The *ScoreCompute* module conducts the *min*() operation based on the match information (i.e. it chooses the proper scores from 9 different options). To further improve the *AlignModule*'s performance, we also insert several registers to shorten the critical path, which introduces a 3 cycle delay. Four bits are applied to represent each alignment score and three bits are applied to represent each start location. The alignment concatenation module design follows the method shown in Figure 6.1 and it also uses a tree structure to reduce the computation latency.

Currently, our design can support a maximum 100bp and a minimum 36bp short read alignment. For reads longer than 100bp, we can simply duplicate the *Align8bp* module to extend the Align Core's align capability. As the read length increases, the *Align Core*'s performance will also drop. This is because more cycles are required to load the (*read*, *ref*) pair. For alignments allowing more gaps, the parallel block alignment algorithm still works except that the search space needs to be expanded. The current *Align Core* only supports alignments with a band width of one. To cope with more gaps, we need to expand our search space by adding more cells into the *Align8bp* module (e.g. if two gaps are allowed, the *Align8bp*

module will require 5×8 cells in total; the connection between each cell is still similar to the one shown in Figure 6.6).

6.2.2 Seed generation design

As shown in Table 5.1, the seed generation stage occupies 13.5% of overall runtime. Initially, it was difficult to decide whether to put this part of the computation on the FPGA or on the CPU. Thus, we apply two different strategies on the seed generation stage design: Seed generation on CPU and Seed generation on FPGA, which leads to different structures on the final hybrid system.

6.2.2.1 Seed generation on CPU

In this strategy, both reference genome indexation and seed generation are computed on the host PC. In our genome indexing design, we choose a similar method to the one used in GASSST: shift the seed window of length k along the reference genome and construct a lookup table based on the coded seed; extra base information, on both sides of the seed, is recorded for the subsequent extension stage. GASSST only saves 16bp extra bases on each side of the seed, which is enough for pre-filter computation. However, these extra bases are not enough for the complete NW search for longer reads (e.g. read length of 76bp) and the missing segments thus need to be retrieved during runtime. Unlike GASSST, our design stores all the necessary data for a complete search during indexing. For example, for a 100bp read search, and a seed length of 19, we store 82bp on each side for possible alignment computation (81bp to achieve 100bp data length, and one extra base to prevent possible sensitivity loss), which will consume about 5 times the memory space than GASSST uses during indexation. One alternative solution to reduce memory consumption is to store less base information and to retrieve the necessary data during the seed generation stage. The price is that more time is required for the seed generation computation. The overall system architecture is shown in Figure 6.9.

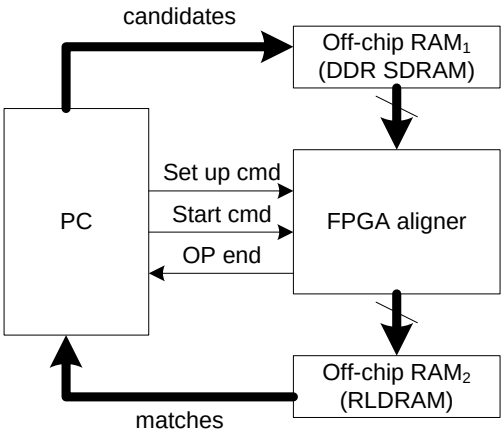


Figure 6.9 Hybrid aligner I - with one FPGA component

Our experimental platform provides two types of off-chip memories for the *Align Core* module access: (1) 2GB of DDR SDRAM with 128bit bandwidth and a maximum clock frequency of 200MHz (referred to as the off-chip RAM_1); (2) 256MB of RLDRAM with 64bit bandwidth and a maximum clock frequency of 200MHz (referred to as the off-chip RAM_2). In practice, each base occupies 2 bits; the position information for the reference fragments is 32 bits; and the read identification number is 32 bits. For example, a 36bp alignment consumes at least 208 bits for the alignment data. This requires four cycles to transfer the data through the off-chip RAM_2 interface, while only two cycles are required for the off-chip RAM_1 interface. Thus, choosing the off-chip RAM_1 to store the candidate data provides more computational benefits. Unlike the (*read*, *ref*) data transfer, the constraint on the alignment results transfer is less stringent, as the number of good alignments is much smaller than the number of input candidate hits. The off-chip RAM_2 provides enough bandwidth for the corresponding data transfer. Predefined commands are sent through the HyperTransport interface to configure *Align Core*'s working modes. Unlike other designs which conduct the alignment read by read, our heterogeneous system computes the alignment chunk by chunk. As the number of candidate hits per read varies, we construct the candidate data chunk based on a fixed number of reads. In this implementation, the host PC generates all candidate hits for 10000 reads as a chunk and sends the associate data to FPGA for alignment. The data chunks are stored in the off-chip RAM_1. When the chunk data is ready,

the host PC issues a *start* command to start the *Align Core*. All match results will be sent to the off-chip RAM_2. When the chunk processing is finished, the *Align Core* sends back an *operation_done* flag to inform the host PC. After that, the host PC will read out the match information from off-chip RAM_2.

As our final system is a hybrid design including a general purpose CPU and an FPGA, a processing speed difference will appear between them. To maximize the FPGA's computation capability, multi-threading is supported on the host PC (i.e. the seed generation stage is multi-threaded). Since the FPGA only has a fixed interface for data transfer, different threads have to share the same FPGA *Align Core*. The scheduling among different threads is controlled by the operating system. The basic scheduling rule is based on priority: once a thread generates all the candidate hits and the *Align Core* is in idle state, it will get the *Align Core*'s processing priority. The thread will keep the priority unless computation is completed. Meanwhile, other threads need to wait for priority after they generate the necessary data. If *Align Core* works much faster than seed computation on a single thread, the overall computation time can be further reduced.

6.2.2.2 Seed generation on FPGA

The seed generation stage design in the previous section includes two computation steps: read coding and match information retrieving. The original read dataset is represented using alphabet {*A*, *C*, *G*, *T*}. Before conducting the similarity search, we should first code each read with a binary format (2bits for each base). The match information retrieving step needs to access a pre-generated hash table and returns the reference segment and the corresponding position in the reference genome. As there are multiple off-chip memories attached to the FPGA device, we decide to store the position information and the reference genome into different memories to pipeline the access. The overall structure for our Hybrid aligner is shown in Figure 6.10.

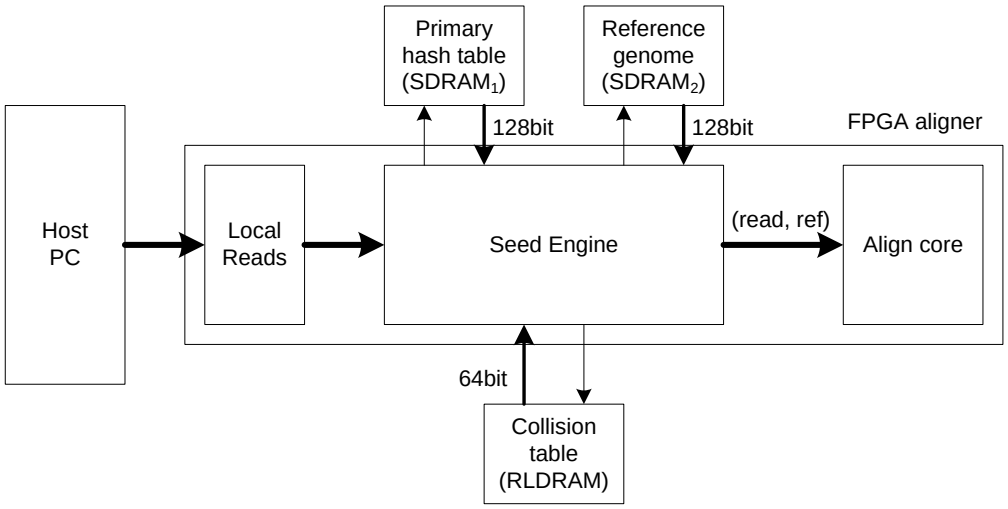


Figure 6.10 The hybrid aligner II structure

Similarly, the read data is also processed chunk by chunk. Unlike the CPU-based design, we store one chunk of reads using on-chip BRAMs (local reads) to permit fast data access. To reduce the BRAM consumption, the chunk size is set to one thousand. When the chunk data processing is finished, a new chunk of data is loaded into the same BRAM space. As the read coding step converts characters into binary strings, we still have to compute this step using a software program. In our design, the information retrieving step is put on the FPGA device (the *Seed Engine* module in Figure 6.10).

The core of the *Seed Engine* design is the hash table construction. Software tools (e.g. GASSST) usually apply a flexible seed model, i.e. for different read lengths using different seed sizes. The advantage of the flexible seed model is that it can avoid generating too many candidate regions and alleviate the work burden for the extension stage. However, the flexible seed model also requires repetitive reference genome indexations. The hash table contents will be updated, if the read length changes. This naturally introduces extra computation effort. In our *Seed Engine* design, the constraint on the number of candidate regions is less stringent, as the *Align Core* is expected to provide much faster alignment processing speed. Therefore, a much simpler constant-length seed model is used. The reference genome only needs to be processed once and the results can be used for different

read datasets. Thus, the indexing is treated as a pre-processing step, which is not included in the overall alignment work flow. To evaluate the influence of different seed models, we record the number of duplicate seeds and alignments with different seed length settings in Table 6.1 (using the GASSST software implementations with one million simulated 76bp reads against the *E. Coli* genome).

Table 6.1 Number of duplications and alignments for different seed lengths

Seed length	**# of duplications**	**# of alignments**
12bp	1,260,817	1,076,671
13bp	504,104	1,049,721
14bp	217,461	1,029,516
15bp	124,197	1,028,288

Table 6.1 shows that a longer seed length can effectively reduce the number of duplications, but the number of alignments reported also drops correspondingly. As duplicate seeds require extra computation time for hash table access, we arbitrarily choose 15bp as the seed length to avoid a large number of duplications while slightly sacrificing the alignment sensitivity. To quickly identify candidate regions, two requirements exist for the hash function selection: (1) a simple hash function representation, and (2) reduced collision generation. A complex hash function would compete with the hardware resources needed for the subsequent extension stage. On the other hand, if there are too many collisions, the additional off-chip memory access will reduce the overall performance.

We apply the bucket hash strategy with hash functions chosen from the H_3 family [106]. The bucket hash first divides the seeds into multiple buckets. A separate hash function is applied for individual buckets to guarantee that there are only a limited number of hash collisions. A seed is divided into two parts: the prefix and the suffix. Figure 6.11 gives an example of the hash query data path for a 15bp seed. As the DNA alphabet size is only four characters, 30bits is enough to represent a 15bp seed. The first 16bits are used as the prefix to construct the buckets; the remaining 14bits (suffix) are used for hash computation. The hash

table address is computed by combining both the prefix and the hash value. As collisions only appear among seeds in the same bucket, the bucket hash strategy can effectively reduce the number of collisions.

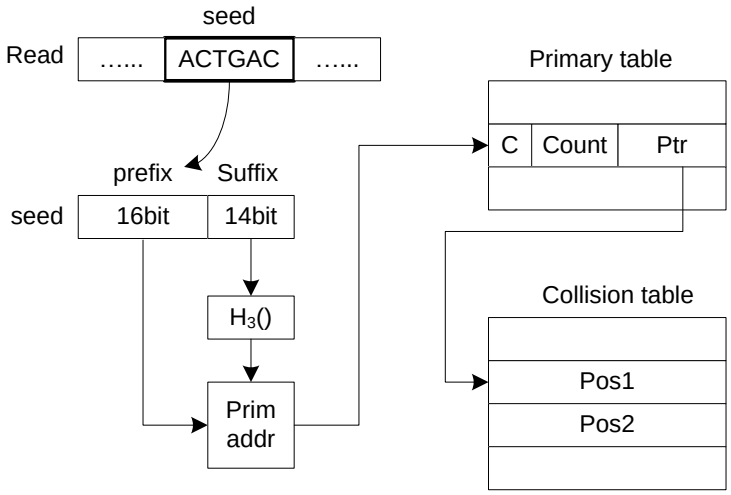


Figure 6.11 Hash query data path

To avoid sensitivity loss for seeds which collide, an additional table, called the collision table, is also constructed. The definition of the primary table and the collision table fields are:

C 1bit, the collision flag

Count 7bits, the total number of seeds that were hashed to this slot

Ptr 24bits, if the collision flag is '0', it indicates the position in the reference genome; otherwise, it indicates the start location in the collision table

Pos 24bits, the match position in the reference genome

For the *E. Coli* reference genome, the primary table consumes 64K×4K×32 bits of memory and the collision table consumes 421623×32 bits of memory. As both the primary hash table and the collision table are too large for the on-chip BRAMs, the primary hash table is stored in DDR2 $SDRAM_1$ and the collision table is stored in RLDRAM (as shown in Figure 6.10). Storing them in separate memories allows us to pipeline the hash query rather than waiting for a complete query operation. Once

a match position in the reference genome is found, the *Seed Engine* will extract the related base information from $SDRAM_2$ (the reference genome) to generate the valid (*read*, *ref*) pair for the extension stage. As the *Seed Engine* computation refers to data transfer between the FPGA and the off-chip memories. Its performance is largely influenced by these I/O operations. The estimated highest clock frequency for the Seed Engine design is around 250MHz (reported by the Xilinx synthesis tool).

In addition to modifications on the *Seed Engine* design, we also integrate a new function into the hybrid system. Unlike the hybrid aligner I system design which only supported single-end short read alignments, the hybrid aligner II system can also support paired-end short read alignments. The paired-end alignment problem involves a pair of short read datasets and two parameters, an outer distance D and a standard variation d. Similar to the single-end alignment, each read in the pair (s_1, s_2) will firstly be aligned individually, which can be accomplished utilizing our FPGA-based short read aligner. Define that p_1 is the start location in the reference genome that read s_1 is aligned to; p_2 is the start location in the reference genome that read s_2 is aligned to. Then, an additional pairing scheme is applied to complete the paired-end search on the host PC. The pairing scheme includes three cases: (1) both s_1 and s_2 have matches in the reference genome; only s_1 has matches in the reference genome; (3) only s_2 has matches in the reference genome. The pseudo code for the pairing scheme is shown in Figure 6.12. For each SW(·) operation in the pseudo-code, we conduct the forward and the reverse complement Smith-Waterman alignment of the short read to increase sensitivity.

```
Assuming that p1 is smaller than p2
if both s1 and s2 can be aligned in the reference genome
   for each match position p1 of s1
         for each match position p2 of s2
            if the |p2 − p1 − D| ≤ 3d
                  output the pair and break
else if no match is found
   if s1 has matches
         for each match position p1 of s1
            conduct SW(s2, ref) using reference segment of position p1+D-d
            if the alignment score ≤ threshold
                  output the pair and break
   else if no match is found
         if s2 has matches
            for each match position p2 of s2
               conduct SW(s2, ref) using ref segment of position p2 -D-d
               if the alignment score ≤ threshold
                     output the pair and break
```

Figure 6.12 Pseudo code for the paired-end alignment

6.3 Performance analysis

We have implemented our short read aligner as a hybrid system. The FPGA part is implemented using Verilog HDL and targeted to a Xilinx Virtex5 LX330 device, which operated with a 200MHz clock due to limited choice of clock frequency. The performance of our hybrid system is compared with GASSST (version 1.26) and BWA (version 0.5.9). The fundamental work flow of our hybrid system is similar to that of GASSST, except that GASSST uses multiple pre-filters to eliminate irrelevant data before the computation intensive global alignment. Although BWA uses a totally different strategy (a BWT-based backward search) to conduct the short read alignment computation, it is probably one of the fastest aligners to date for alignments with a low error rate and is also a very popular choice in the bioinformatics community. Thus, we also include BWA in the performance evaluation. Gaps are allowed in all three designs. All tests are performed on a quad-core (each core running at 2.1GHz) AMD Opteron processor with 8Gbyte RAM running the Linux OS. The reference genome is the *E. Coli* NC_008253 dataset with 4.9 million residues. We have generated several simulated short read datasets using the *wgsim* utility program in the SAMtools

package [70] (version 0.1.17) with different error rate settings. In all our experiments, the percentage of identification is set to 90% and the number of gaps allowed is one. For example, for a 36bp read alignment, at most 3 errors are allowed (at most one indel error and the rest are substitutions or single nucleotide polymorphisms (SNPs)). SNP is a special case of base substitution. In our design, we also treat SNP as a substitution error. The other parameters for GASSST and BWA are set to the default values.

As both of our hybrid system designs share the same *Align Core*, it is meaningful to give a separate runtime performance evaluation on this module. We compare the runtime performance between the filtration part in GASSST and our *Align Core*, assuming that the input data for each of them is available. As this experiment only compares the "alignment" section runtime, the runtime performance of BWA is not included. Three small simulated short read datasets (36bp) of varying error rates are used (The error rate is one parameter of the simulation tool SAMtools package. It represents the probability of mutation on a single character in the generated short read dataset). Each dataset contains one million reads. Table 6.2 reports the runtime comparison under different error rate settings. The GASSST filter runtime includes all filtration stages from the tiled-NW filter to the real NW filter computation using a single thread. The *Align Core* runtime also includes data transfer from off-chip memory to FPGA.

Table 6.2 Alignment runtime comparison for one million reads

	Runtimes (s)		
	4% error rate	**6% error rate**	**8% error rate**
GASSST	55	46	37
Align Core	1.6	1.47	1.31
Speedup	34×	31.3×	28×

Table 6.2 shows that the FPGA *Align Core* provides a much better runtime performance on the alignment computation (around 30 times faster than that in GASSST). As the filtration (or extension) stage is accelerated by the *Align Core*, it is no longer the performance bottleneck for the "seed and extend" strategy. Before

the performance analysis on different hybrid aligners, Table 6.3 gives a simple comparison on their resource utilizations. Obviously, hybrid aligner II consumes more resources, as the seed generation stage is implemented on the FPGA.

Table 6.3 Resource utilization for different hybrid aligners

	Hybrid aligner I	**Hybrid aligner II**
Number of slice registers	49,551	81,195
Number of slice LUTs	74,595	78,218
Total memory used (KB)	2,934	4,752

6.3.1 Hybrid aligner I analysis

In hybrid aligner I, the seed generation stage is built as software running on the general-purpose CPU and only the *Align Core* is designed on the FPGA chip. Before we analyze the runtime performance of this hybrid system, we test the alignment quality for different designs. We use two factors to assess the quality of an alignment tool: (i) the total number of alignments found by the tool (sensitivity) and, (ii) the number of true alignments found (accuracy). Since we know the exact positions of each read in the simulated datasets, the correctness of the reported alignments can be easily evaluated by comparing the position information. In practice, if an alignment position is within a maximum distance d (d=5 in all our evaluations) to the correct position, this alignment will be labeled as a correct alignment. Table 6.4 ~ 6.6 show the quality comparison between GASSST, BWA, and the hybrid aligner I using one million simulated reads with various read lengths and error rates.

Table 6.4 Alignment results for one million reads of 36bp

		4% error rate	6% error rate	8% error rate
GASSST	# of align	911,549	783,381	623,104
	# of true align	895,100	769,279	611,690
BWA	# of align	923,874	659,382	485,416
	# of true align	907,488	647,651	476,416
Hybrid aligner I	# of align	912,163	784,263	624,140
	# of true align	895,714	770,164	612,709

Table 6.5 Alignment results for one million reads of 76bp

		4% error rate	6% error rate	8% error rate
GASSST	# of align	966,001	863,848	668,195
	# of true align	951,587	851,162	658,185
BWA	# of align	831,634	643,670	438,981
	# of true align	819,522	634,386	432,518
Hybrid aligner I	# of align	962,314	853,587	655,585
	# of true align	947,422	840,382	645,163

Table 6.6 Alignment results for one million reads of 100bp

		4% error rate	6% error rate	8% error rate
GASSST	# of align	976,574	905,030	728,123
	# of true align	963,265	892,342	717,972
BWA	# of align	835,099	655,336	454,092
	# of true align	814,047	646,444	447,858
Hybrid aligner I	# of align	980,917	918,196	765,215
	# of true align	963,954	901,372	750,350

The alignment results show that BWA's sensitivity and accuracy drop dramatically compared to GASSST and our FPGA aligner, when the base error rate increases. In contrast, both GASSST and the hybrid aligner I maintain the performance within a relatively stable level: over 78% sensitivity for 6% error rate and over 60% sensitivity for 8% base error rate. All three designs provide over 98% alignment accuracy. Generally, the hybrid aligner I has a similar sensitivity and accuracy to

that of GASSST under these test conditions. To get a more comprehensive evaluation on three aligners, we also test their performances using larger datasets (20 million sequences of length 36bp, 76bp, and 100bp, respectively; the base error rate is set to 4%). The alignment quality is recorded in Table 6.7 and the runtime performance is recorded in Table 6.8.

Table 6.7 Alignment results for 20 million reads with 4% base error rate

		36bp	**76bp**	**100bp**
GASSST	# of align	18,233,687	19,318,463	19,524,380
	# of true align	17,907,896	19,034,347	19,255,520
BWA	# of align	16,627,236	16,627,675	16,687,423
	# of true align	16,331,206	16,387,736	16,464,717
Hybrid aligner I	# of align	18,245,101	19,242,887	19,614,500
	# of true align	17,917,304	18,947,439	19,277,337

Table 6.8 Execution time comparison among different parameter settings

Execution time for 36bp reads under different thread settings			
	1 thread	**2 threads**	**4 threads**
GASSST	35 min 37 sec	18 min 12 sec	13 min 26 sec
BWA	21 min 54 sec	11 min 26 sec	7 min 57 sec
Hybrid aligner I	13 min 41 sec	8 min 17 sec	5 min 32 sec
$Speedup_1$	2.6×	2.2×	2.43×
$Speedup_2$	1.6×	1.38×	1.43×
Execution time for 76bp reads under different thread settings			
	1 thread	**2 threads**	**4 threads**
GASSST	1 hr 7 min 14 sec	35 min 26 sec	23 min 2 sec
BWA	50 min 48 sec	25 min	17 min 59 sec
Hybrid aligner I	23 min 35 sec	15 min 50 sec	10 min 17 sec
$Speedup_1$	2.63×	2.24×	2.24×
$Speedup_2$	1.98×	1.58×	1.75×
Execution time for 100bp reads under different thread settings			
	1 thread	**2 threads**	**4 threads**
GASSST	1 hr 26 min 16 sec	44 min 26 sec	28 min 31 sec
BWA	1 hr 10 min 57 sec	50 min 26 sec	23 min 55 sec
Hybrid aligner I	23 min 35 sec	15 min 50 sec	10 min 17 sec
$Speedup_1$	2.71×	2.32×	2.16×
$Speedup_2$	2.23×	2.63×	1.81×

($Speedup_1$ indicates the runtime comparison between the hybrid aligner I and GASSST; $speedup_2$ indicates the execution time comparison between hybrid aligner I and BWA).

Table 6.7 shows that BWA only achieves around 83% sensitivity, while GASSST and the hybrid aligner I achieve between 91% and 98% sensitivity, which means that both GASSST and the hybrid aligner I are more effective under this error setting. Since BWA can reuse the indexed reference genome for different read alignments, the execution time for BWA represents only the alignment computation part. In contrast, the execution time for GASSST and the hybrid aligner I includes reference indexing and read alignment computation. In this experiment, BWA provides faster processing speed than GASSST by taking advantage of the BWT-based search. The hybrid aligner I provides an average

speedup of 2.4 over GASSST and of 1.8 over BWA. Notice that the performance of the hybrid aligner I reported in Table 6.8 is partially multi-threaded, i.e. different threads share the same *Align Core* which only processes data from one thread at a time. One interesting observation is that no matter how many threads are applied, the hybrid aligner I achieves around 2 times speedup over GASSST. As *Align Core* alone can achieve over 30 times speedup against the filtration section in GASSST at this base error rate setting, it is clear that the processing capability of the *Align Core* is not fully utilized. This is because the software-based seed generation stage design cannot generate enough (*read*, *ref*) pairs to feed the *Align Core*. The general purpose CPU pre-caches part of the data from the main memory to accelerate consecutive memory accesses. Unfortunately, this mechanism makes the performance deteriorate significantly for random memory accesses. Therefore, the seed generation stage becomes the performance bottleneck after we accelerate the alignment computation. As our indexing strategy is slightly different from that of GASSST, the indexing runtime performances for two designs are also evaluated on Table 6.9.

Table 6.9 Indexing time comparison for different read lengths

	Runtime (s)		
	36bp	**76bp**	**100bp**
GASSST	3.51	7	7
Hybrid aligner I	4.93	8.44	8.44

Although the indexing time required by the hybrid aligner I is slightly higher than that of GASSST, compared to the overall execution time, such influence is negligible. In general, the core of our design, the *Align Core*, provides competitive performance improvements to accelerate the most time consuming part of the short read alignment and also maintains a high degree of sensitivity and accuracy compared to the two other widely-used alignment tools.

6.3.2 Hybrid aligner II analysis

In this version of our hybrid short read aligner design, we map the seed generation stage into the FPGA chip to further accelerate the computation process. Similar to the analyses made in 6.3.1, we first evaluate the execution time of GASSST, BWA, and the hybrid aligner II using one million reads with 4% error rate with different read lengths (36bp, 76bp, and 100bp, respectively). The experiment is also conducted under multi-thread conditions (shown in Table 6.10).

Table 6.10 Execution time comparison among different thread conditions

Execution time for 36bp read dataset (s)			
	1 thread	**2 threads**	**4 threads**
GASSST	110	58	42
BWA	64	34	22
Hybrid aligner II	16	11	11
$Speedup_1$	6.8×	5.3×	3.8×
$Speedup_2$	4.0×	3.1×	2.0×
Execution time for 76bp read dataset (s)			
	1 thread	**2 threads**	**4 threads**
GASSST	216	116	78
BWA	150	88	54
Hybrid aligner II	30	21	21
$Speedup_1$	7.2×	5.5×	3.7×
$Speedup_2$	5.0×	4.2×	2.6×
Execution time for 100bp read dataset (s)			
	1 thread	**2 threads**	**4 threads**
GASSST	268	145	96
BWA	214	104	71
Hybrid aligner II	38	28	28
$Speedup_1$	7.1×	5.2×	3.4×
$Speedup_2$	5.7×	3.7×	2.5×

Similar to the hybrid aligner I, the performance of the hybrid aligner II also improves with partial multi-threading. Table 6.10 shows that the hybrid aligner II

achieves a speedup of 7 over GASSST and of 5 over BWA with a single thread. Although the performance is much better than that of hybrid aligner I, the speedup is still less than that of *Align Core* alone. When alignment computation is no more the performance bottleneck, two other factors are responsible for the system's overall performance: the read coding and the Seed Engine processing. The read coding computation is simple, but the total number of bases is large. Therefore, we utilize Streaming SIMD Extensions 2 (SSE2) instructions to minimize its computation time. Notice that there is no performance improvement for the hybrid aligner II, when the number of threads changes from two to four. This indicates that system's performance might be limited by the shared FPGA aligner. To prove this, we further record the runtime consumed by the FPGA aligner under different thread conditions (shown in Table 6.11).

Table 6.11 The runtime composition for one million 100bp reads

	Runtime (s)		
	1 thread	**2 threads**	**4 threads**
Total Execution time	38	28	28
FPGA aligner runtime	13.4	21.65	21.57
Percentage	35.2%	77%	77%

When two threads are applied, the FPGA aligner's runtime is almost doubled and the percentage jumps from 35.2% directly to 77%. As the FPGA aligner is shared by two threads, it is reasonable to have such an observation. However, since the FPGA aligner becomes the dominant factor for the system's overall runtime using two threads, it is hard to expect further performance improvements when more threads are applied, as can be seen from the four thread experiment. As the FPGA aligner consists of two major components (the *Seed Engine* and the *Align Core*), and the *Align Core* is no longer the performance bottleneck, the only explanation for the performance limitation is the *Seed Engine*. Although the FPGA-based *Seed Engine* provides a higher band width (128bit bandwidth) and avoids the useless data pre-fetching for each hash table query, the intensive random off-chip memory access is still the performance bottleneck for the hybrid aligner II design. The Seed

Engine computation requires frequent off-chip memory accesses (multiple cycles are required for a complete off-chip memory access), which intrinsically limits its runtime performance. Several methods exist to mitigate the negative influence of off-chip hash table access: (1) using FPGA platforms with multiple off-chip memory interfaces (thus, allowing more hash lookups at the same clock cycle), or (2) using FPGA chips with larger on-chip memories (the collision table can then be stored using BRAM; thus a single off-chip access will be enough for a hash query). However, as the off-chip memory access is always slower than on-chip memory access and the hash table size is large, it is still difficult to get a great performance improvement on the overall execution time.

Runtime testing with a single error rate might not be enough to evaluate the performance of the hybrid aligner II. We also test the runtime performance under different error rate conditions using simulated read datasets with a single thread. The performance is shown in Table 6.12.

Table 6.12 Runtime performance under different error rate conditions

76bp read dataset (s)			
	2% error rate	**4% error rate**	**6% error rate**
GASSST	227	216	199
BWA	109	150	162
Hybrid aligner II	30	30	29
$Speedup_1$	7.6×	7.2×	6.8×
$Speedup_2$	3.6×	5.0×	5.6×
100bp read dataset (s)			
	2% error rate	**4% error rate**	**6% error rate**
GASSST	292	268	253
BWA	153	214	221
Hybrid aligner II	37	38	37
$Speedup_1$	7.9×	7.1×	6.8×
$Speedup_2$	4.1×	5.7×	5.9×

From Table 6.12, we see that more execution time is required for all three alignment tools, as a longer read length is applied. This is because a longer read length leads to a larger search space, which naturally requires more computation time. Another interesting observation is that, under different error rate conditions, the hybrid aligner II provides a relatively stable performance. In contrast, GASSST shows performance improvements and BWA shows performance deterioration, as the error rate increases. The search strategies applied within the tools can explain such a difference. BWA applies the BWT-based search algorithm, which consumes additional computation time when an error occurs. Therefore, its computation time will increase, if more errors exist in the read dataset. The GASSST computation is based on the "seed and extension" strategy. As the error rate increases, the number of seeds in the read decreases, which reduces the computation time required in the extension stage. The hybrid aligner II also uses the "seed and extension" strategy, but the FPGA aligner computation only occupies a small portion of the overall runtime under single thread condition. Thus, the error rate change only has limited influence on the execution time. Besides investigating the processing speed, we also record the alignment quality under different test conditions, as shown in Table 6.13.

The alignment results show that BWA's sensitivity drops dramatically compared to GASSST and the hybrid aligner II, when the base error rate increases. In contrast, both GASSST and the hybrid aligner II maintain a relatively stable performance: over 96% sensitivity for a 4% error rate and over 85% sensitivity for a 6% error rate. All three tools provide over 98% alignment accuracy. The alignment quality of the hybrid aligner II is slightly better than that of GASSST under above test conditions. The reason for the improvement is related to the seed model. GASSST uses a flexible seed model to ease the alignment computation. For longer read lengths, GASSST will use longer seed lengths under the default setting (e.g. a 16bp seed for the 76bp read dataset; and a 19bp seed for the 100bp read dataset), which is a trade-off between processing speed and sensitivity. In contrast, hybrid aligner II uses a constant seed model (a 15bp seed length), which thus provides extra sensitivity for longer read datasets.

Table 6.13 Alignment results for different read lengths and error rates

		2% error rate		
		GASSST	**BWA**	**Hybrid aligner II**
76bp	# of align	990,842	964,053	993,057
	# of true align	976,482	950,293	977,722
100bp	# of align	990,970	964,005	993,701
	# of true align	977,781	951,425	979,536
		4% error rate		
		GASSST	**BWA**	**Hybrid aligner II**
76bp	# of align	966,001	831,634	971,367
	# of true align	951,587	819,522	956,785
100bp	# of align	976,574	835,099	988,114
	# of true align	963,265	823,850	974,243
		6% error rate		
		GASSST	**BWA**	**Hybrid aligner II**
76bp	# of align	863,848	643,670	873,804
	# of true align	851,162	634,386	861,025
100bp	# of align	905,030	655,336	944,693
	# of true align	892,342	646,444	931,325

We have also conducted a further performance comparison of GASSST, BWA, and the hybrid aligner II using larger read datasets. The simulated read datasets are 20 million sequences of length 76bp and 100bp, with a 4% error rate. The alignment performance is shown in Table 6.14 for a single thread implementation.

Table 6.14 Alignment performance for 20 million read datasets

4% error rate 76bp			
	# of align	**# of true align**	**Execution time**
GASSST	19,318,463	19,034,347	67 min 14 sec
BWA	16,627,675	16,387,736	50 min 48 sec
Hybrid aligner II	19,424,609	19,137,326	9 min 54 sec
4% error rate 100bp			
	# of align	**# of true align**	**Execution time**
GASSST	19,524,380	19,255,520	86 min 16 sec
BWA	16,687,423	16,464,717	70 min 57 sec
Hybrid aligner II	19,760,231	19,484,102	12 min 32 sec

In this experiment, BWA only achieves around 83% sensitivity, while GASSST and the hybrid aligner II achieve around 97% sensitivity. This indicates that both GASSST and the hybrid aligner II are more effective in finding good alignments under this error rate setting. On the runtime performance side, BWA provides a faster processing speed than GASSST. The hybrid aligner II achieves the fastest processing speed, while maintaining the sensitivity at a high level. In addition to the simulated short read datasets, we have also evaluated the alignment performance using the real dataset (short read sample from SRR519953, N's are replaced with a random character, in total 1,884,895 paired reads). The original read length within SRR519953 is 101bp. As our design only supports a maximum 100bp short reads, we chop one character off from the 3'end. For the single-end alignment, we use first 1,888,895 short reads to conduct the test. Table 6.15 records the alignment quality and the runtime performance with a single thread for three aligners. As the correct position for an alignment is unknown for the real dataset, we only use the number of alignment reads or aligned rate to evaluate the aligner's performance. It is clear that hybrid aligner II reports more alignments than the other two aligners with the fastest processing speed (over 6 times speedup).

Table 6.15 Single-end short read alignment using real dataset SRR519953

	# of alignment	Aligned rate	Execution time
GASSST	1,477,911	78.4%	467 sec
BWA	1,472,773	78.1%	444 sec
Hybrid aligner II	1,522,697	80.7%	65 sec

As GASSST does not support paired-end alignment, the performance comparison is made only between BWA and the hybrid aligner II using one million simulated short reads. Before the performance analysis, we further introduce several parameters to evaluate the alignment quality: precision (defined as the number of true alignments/ total alignments), recall (the number of true alignments/ the number of reads), and F-score (F-score $= \frac{2\times precision\times recall}{precision+recall}$). The F-score combines the influence of precision and recall together to evaluate the alignment quality. The comparison results are shown in Table 6.16. The paired-end read dataset is generated using *wgsim* utilities with an outer distance of 200, a standard variation of 20, and 4% error rate. The performance is tested with a single thread. Table 6.17 records the performance tested on the real dataset SRR519953.

Table 6.16 Paired-end alignment performance

100bp simulated reads					
	Aligned	**Precision**	**Recall**	**F-score**	**Execution Time (s)**
BWA	1,940,132	99%	96.03%	97.49%	548
Hybrid aligner II	1,962,404	97.9%	96.05%	96.97%	108

Table 6.17 Paired-end alignment with real dataset

	# of alignments	Aligned rate	Execution time (s)
BWA	3,002,675	79.6%	1037
Hybrid aligner	2,841,262	75.3%	300

Table 6.16 results show that hybrid aligner II finds out more paired-end alignments than BWA, but with a slightly worse performance on precision for one million simulated 100bp reads. We also notice that the speedup achieved for paired-end alignment is smaller that of single-end alignment. These observations are related to the pairing strategy applied by BWA which further utilizes the advantage of BWT index to accelerate the pairing process. In Table 6.16, the hybrid aligner II reports more alignments, but BWA reports more alignments in Table 6.17. We believe such differences are also related to the pairing strategy. The reported alignments can be increased simply by increasing the search space ($O(l_1 l_2)$, l_1 is the read length; l_2 is the candidate region) in the pairing process, but the execution time will also increase correspondingly.

The above experiments show that hybrid aligner II achieves a significant performance improvement when testing with a small reference genome (the *E. Coli* reference genome). While the hybrid system design is independent of the reference genome, the restricted memory size of the experiment platform (and not the implementation) limits the possibility to test with a larger reference genome, such as the human genome (with over three billion bases). If no partitioning is applied for the human genome and we use the minimal perfect hash function to construct the hash table, the system requires at least 12GBytes of memory for the hash table (three billion entries for the table and each entry points to an 32bit table content). In contrast, our experimental platform only has 2GByte of off-chip memory for the hash table, which is not large enough to support the human genome. In practice, the memory footprint is likely to even larger than 12GByte, as it is hard to get a minimal perfect hash function for the human genome. A different FPGA platform with more off-chip memory could support the short read mapping against the human genome using our proposed architecture. Another possible solution to support human genome is to apply BWT-based methods in the seed generation stage design to reduce the memory consumption. Utilizing multiple FPGA boards can also ease the reference genome size problem. In fact, [99] is a successful example of conducting the short read alignment against the human genome using 8 FPGAs and 32GB of off-chip memory.

When comparing the performance of hybrid aligner II and hybrid aligner I, we find that hybrid aligner II outperforms hybrid aligner I with better alignment quality and processing speed under the various test settings. The constant seed model and the mapping of the seed generation stage to the FPGA do have obvious improvements on the design's performance. However, both hybrid aligner II and hybrid aligner I still share the same performance bottleneck - intensive random memory access in a large memory space. To provide multiple copies of the hash table on the off-chip memory can get a performance boost, but with a very large memory consumption. Since not all query seeds will be hashed to effective locations, it might be a more memory-efficient solution to add a pre-filter with much less memory consumption, which is also part of our future investigation.

Already we have seen short read aligners built on other hybrid platforms, such as GPUs. CUSHAW achieves around 10 times speedup comparing to BWA (1GPU against 1CPU) tested with short reads from the human genome. Notice that the GPU used in CUSHAW is the NVIDA Tesla C2050 (released around the end of 2009), which provides 3GByte of GDDR5 memory for each GPU [122]. In contrast, our hybrid platform is at least one technology generation older than the Tesla GPU (the LX330 FPGA was released in 2006) and only provides 2GB DDR2 memory for hash table access. We can expect instant performance improvements once the design is migrated to a newer FPGA platform.

6.4 Summary

In this chapter, we have presented two hybrid systems to accelerate the mapping of short reads to a reference genome based on the "seed and extend" approach. We have proposed a banded parallel semi-global alignment architecture to accelerate the computation-intensive extension stage on an FPGA. Meanwhile, we also explore different mapping strategies on the seed generation stage design. When mapping the seed generation stage onto the FPGA chip, we apply a bucket hash structure to improve the seed generation computation. Multiple experiments are conducted to evaluate the hybrid systems' processing speed and alignment quality.

The performance comparison to the GASSST and BWA software implementations shows that both the hybrid aligner I and the hybrid aligner II achieve a high degree of sensitivity and require less overall execution time with only modest resource utilization. As a result, the performance bottleneck on the two hybrid systems changes from the extension stage to the seed generation stage.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have investigated the possibility of applying FPGAs to improve the computation performance of different problems in bioinformatics. As a case study, two important research areas in bioinformatics are chosen, genome search and short read mapping, both of which involve large amounts of data and complicated computations. As genetic datasets are dramatically increasing in size, this has placed a heavy burden on modern CPU-based toolsets. Thus, there is an urgent need for new solutions with higher processing speed, but with equivalent accuracy, to the existing tools in these (and other) bioinformatics fields. FPGAs have been shown to be competitive candidates for high performance computing in other domains, and thus are likely to provide an alternative solution for existing bioinformatics problems. Problems involving intensive computation, weak/no data dependency among operations and parallel data processing can be easily optimized utilizing FPGA platforms. However, problems involving complex control logic or intensive memory accesses might not be perfect for FPGA implementations. To better utilize the FPGA's computation capability, a heterogeneous architecture (i.e. CPU+FPGA) is proposed to meet the different algorithm requirements. Usually, the hybrid design involves three steps: profiling, partitioning, and implementation. The profiling step first identifies the most time consuming part in the algorithm. Then, the partitioning step maps the computation-intensive operations to the FPGA(s) and the complicated control to the general purpose CPUs. Lastly, the implementation step constructs an appropriate high performance architecture on the FPGA and connects the FPGA design to the CPU using standard high-speed interfaces. In this thesis, our algorithm implementations adopt this procedure.

For genome search, the target algorithm is NCBI BLASTN which is a bio-sequence search tool of high importance to bioinformatics research. Among three operation stages in NCBI BLASTN, the word matching stage is the most time-consuming part. Thus, accelerating this stage's performance can achieve the highest performance improvement. Our design includes two innovative structures, a highly efficient parallel Bloom filter and an optimized off-chip block hash table. The Bloom filter design can process 16 data words in parallel with high search efficiency. The comparison of the performance of our word-matching accelerator to that of NCBI BLASTN shows over 10 times speedup against the single-thread NCBI BLASTN program.

For the short read alignment, the performance profiling shows that the extension operation is the most time-consuming part. Thus, we map this stage onto the FPGA. As the short read alignment problem also involves other operations, e.g. read coding, seed generation, and alignment output, it is difficult to determine the optimal partition of these operations in the first place. Two hybrid systems with slightly different partitioning strategies are presented to accelerate the mapping of short reads to a reference genome based on the "seed and extend" approach. We propose a parallel banded semi-global alignment structure to accelerate the extension stage supporting reads of different lengths. In one of our hybrid systems, a bucket hash structure is applied to improve the seed generation stage computation. The performance comparison to the GASSST and BWA software implementations under different test conditions shows that our FPGA aligner achieves a high degree of sensitivity and over 30 times runtime performance improvement.

Although the word matching accelerator and the short read aligner target to different algorithms, they disclose similar design challenges that FPGAs should face for problems with large volumes of data. FPGAs only have a relatively limited on-chip memory resource (several million bits in the majority of cases), which narrows the options for large lookup table construction (or reference genome indexation) and inherently limits FPGAs' computation power. One key advantage of FPGAs is that we can duplicate multiple modules to expand the data processing

capability. However, this can also introduce a new dilemma: the I/O bottleneck. Both the number of physical interfaces and the data transfer rate are not enough for frequent data communication with a large dataset. The duplicated resources will be in an idle state, if there is not enough input data. Another possible challenge is the computing limitation for a single FPGA. Recall that bioinformatics problems involve a large volume of independent data. A system with multiple FPGAs might be a more competitive candidate in this particular field.

7.2 Future work

Despite the above challenges facing FPGAs, existing FPGA-based designs exhibit high performance for parallel computing and fine-grained pipelining. The architecture optimization is probably one of the most attractive and important parts for an FPGA design, which determines the computation efficiency and overall performance. Different from other fields, bioinformatics algorithms usually apply heuristics to accomplish a complicated task. This provides us the freedom to propose FPGA-favorable heuristics or to tailor existing algorithms for FPGA implementation. These opportunities expand the space for FPGA design exploration. Thus, case studies are a proper choice for these design concept evaluations.

The word matching stage design for the genome search problem is only the first stage of the NCBI BLASTN algorithm. Although the ungapped extension stage consumes less computation than the word matching stage, its independent data structure makes it very suitable for parallel implementations on an FPGA.

In the short read mapping problem, we have proposed two hybrid system structures to accelerate its computation. However, the issue of intensive off-chip memory access is still waiting for a better solution. Meanwhile, the memory footprint for the "seed and extend" strategy is large. If it is possible to integrate the BWT-based search algorithm into our seed generation stage design, further performance improvements can be expected.

Both FPGAs and GPUs are competitive platforms for bioinformatics applications, but it is difficult to proclaim that FPGAs outperform GPUs in all circumstances. In fact, FPGAs and GPUs have advantages on different types of applications. GPUs can achieve good performance for applications involving the same operations executed on a large volume of data. Hundreds of CUDA cores can provide an order of magnitude performance improvement compared to general purpose processors. In contrast, the advantage of FPGAs lies in their fine-grained pipelining and massive parallelism. FPGAs can provide the freedom to fully customize the circuit to fit a specific application, and are especially efficient for fixed point, integer, or bit operations. However, as many applications contain complicated algorithms, an architecture merely relying on GPUs or FPGAs cannot achieve optimal performance. Heterogeneous systems consisting of CPU, GPU and FPGA can further expand a design's application scope and improve the performance and so is worthy of further study. To optimize the performance of a heterogeneous system, data flow balancing among different computing platforms is usually required, which can be an interesting direction for our future investigations.

Fortunately, bioinformatics can provide a wide range of algorithms to examine the influence of different system-level optimizations or tradeoffs on a heterogeneous system. For example, the phylogenetic tree can be another interesting algorithm suitable for heterogeneous implementation. A Phylogenetic tree is a branching diagram that reveals the evolutionary relationship among different species. Normally, three types of methods exist to construct a tree: distance-based method, parsimony-based method, and maximum likelihood-based method. The distance-based methods (e.g. UPGMA [106] and NJ [103]) reconstruct the tree utilizing the evolutionary distance matrix. Parsimony-based methods reconstruct the tree with the constraint of minimizing the number of changes. The maximum likelihood-based methods reconstruct the tree with the constraint of maximizing the likelihood of the data. The maximum likelihood methods are computation expensive. Normally, heuristics are applied to reduce the computation cost. Examples of tools based on maximum likelihood methods are dnaML [39], fastdnaML [100], RAxML [118,119], and so on. Although the heuristics can reduce the computation

time, it still requires a lot of time on the repetitive computation of the likelihood function for each possible insertion point. Thus, it is possible to map the computation intensive operations onto an FPGA and construct a heterogeneous system to further reduce the tree reconstruction time.

De novo short read assembly is another interesting and important field in next generation sequencing that might be suitable for FPGA implementation. Software tools, such as Velvet [135], AByss [113], SOAPdenovo [75], etc, have been developed to solve the short read assembly problem. Two main bottlenecks exist with these tools: (1) the computation is too slow, and (2) they require a tremendous memory space, which largely limits their application. Hybrid systems (i.e. CPU+GPU/ CPU+FPGA) can be an alternative to solve this kind of problem in a more efficient manner. Already we have seen hybrid designs (e.g. PASHA [80], [83]) that have been proposed to accelerate the assembly process. Since the memory footprint for the complete genome is large, the common solution is to split the genome into smaller chunks and assign the data chunks to the accelerators. FPGAs have demonstrated their outstanding capability on computation intensive operations, but the limited memory resources can be a potential bottleneck. If it is possible to further assign the memory access operations and similarity computation operations to different platforms (e.g. GPU+FPGA), an order of magnitude performance boost of the assembly process can be expected.

Bibliography

[1] Abouelhoda MI, Kurtz S, and Ohlebusch E, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, 2(1):53-86, 2004

[2] Abouellail M, El-Araby E, Taher M, El-Ghazawi T, and Newby GB, "DNA and protein sequence alignment with high performance reconfigurable systems," Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 334-341, 2007

[3] Aho A, and Corasick MJ, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, 18(6): 333-340, 1975

[4] Alachiotis N, Berger S, and Stamatakis A, "Accelerating phylogeny-aware short DNA read alignment with FPGAs," IEEE 19th annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 226-233, 2011

[5] Alachiotis N, Sotiriades E, Dollas A, and Stamatakis A, "Exploring FPGAs for accelerating the phylogenetic likelihood function," IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 1-8, 2009

[6] Alachiotis N, Stamatakis A, "FPGA acceleration of the phylogenetic parsimony kernel," International Conference on Field Programmable Logic and Applications (FPL), 417-422, 2011

[7] Altschul SF, Gish W, Miller W, Myers EW, and Lipman DJ, "Basic local alignment search tool," Journal of Molecular Biology, 215: 403-410, 1990

[8] Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, and Lipman DJ, "Gappped BLAST and PSI-BLAST: A new generation of protein database search program," Nucleic Acids Research, 25: 3389-3402, 1997

[9] Amdal GM, "Validity of the single processor approach to achieving large-scale computing capabilities," AFIPS Conference proceedings (30) : 483-485, 1967

[10] Anderson S, "Shotgun DNA sequencing using cloned DBase I-generated fragments," Nucleic Acids Research, 9(13): 3015-3027, 1981

[11] Arlazarov VL, Dinic EA, Kronrod MA, and Faradjev IA, "On economical finding of transitive closure of a graph," Soviet mathematics Doklady, 11: 1270-1272, 1970

[12] Bakos JD, Elenis PE, and Tang J, "FPGA acceleration of phylogeny reconstruction for whole genome data," Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering (BIBE), 888-895, 2007

[13] Benkrid K, Liu Y, and Benkrid A, "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment," IEEE Transactions on VLSI systems, 17(4): 561-570, 2009

[14] Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Wheeler DL, "GenBank". Nucleic Acids Res. 36(Database issue): D25-30, Jan 2008

[15] Berger S, and Stamatakis A, "Aligning short reads to reference alignments and reference trees," Bioinformatics, 27(15): 2068-2075, 2011

[16] Blom J, Jakobi T, Doppmeier D, Jaenicke S, Kalinowski J, Stoye J, and Goesmann A, "Exact and complete short read alignment to microbial genomes using GPU programming," Bioinformatics, 27(10):1351-1358, 2011

[17] Bloom B, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, 13(7): 422-426, 1970

[18] Botelho FC, Kohayakawa Y, and Ziviani N, "An approach for minimal perfect hash functions for very large databases," Technical Report, Department of Computer Science, University Federal de Minas Gerais, Belo Horizonte, Brazil, 2006

[19] Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, and Storaasli OO, "State-of-the-art in heterogeneous computing," Scientific Programming, 18(1): 1-33, 2010

[20] Buhler J, "Mercury BLAST disctionaries: analysis and performance measurement," Technical Report, Department of Computer Science & Engineering, Washington University in St. Louis, 2007

[21] Burrows M, and Wheeler DJ, "A block sorting lossless data compression algorithm," Technical Report 124, Palo Alto, CA: Digital Equipment Corporation, 1994

[22] Carter L, and Wegman M, "Universal classes of hashing functions," Journal of Computer and System Sciences, 18(2): 143-154, 1979

[23] Chazelle B, Kilian J, Rubinfeld, and Tal A, "The Bloomier filter: an efficient data structure for static support lookup tables," Proceedings of the 15th annual ACM-SIAM symposium on Discrete Algorithms (SODA), 30-39, 2004

[24] Chen Y, Souaiaia T, Chen T, "PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds," Bioinformatics, 25(19): 2514-2521, 2009

[25] Cheng L, Cheung DW, and Yiu S, "Approximate string matching in DNA sequences," Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA), 303, 2003

[26] Chrysos G, Sotiriades E, Papaefstathiou I, and Dollas A, "A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM),"

International Conference on Field Programmable Logic and Applications (FPL), 683-686, 2009

[27] Convey Computer, http://www.conveycomputer.com

[28] IBM Cell Broadband Engine, http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

[29] IBM Cell broadband engine programming tutorial version 2.0, http://moss.csc.ncsu.edu/~mueller/cluster/ps3/CBE_Tutorial_v2.0_15December2006.pdf

[30] Dayhoff MO, Schwartz R, and Orcutt BC, "A model of evolutionary change in proteins," Atlas of protein sequence and structure, Vol. 5, No. suppl 3: 345-351, 1978

[31] Delaney S, Butler G, Lam C, and Thiel L, "Three improvements to the BLASTP search of genome databases," Proceedings of 12th International Conference on Scientific and Statistical Database Management, 14-24, 2000

[32] Delcher AL, Kasif S, Fleischmann RD, Peterson J, White O, and Salzberg SL, "Alignment of whole genomes," Nucleic Acids Research, 27(11), 2369-2376, 1999

[33] Delcher AL, Harmon D, Kasif S, White O, and Salzberg SL, "Improved microbial gene identification with GLIMMER," Nucleic Acids Research, 27(23): 4636-4641, 1999

[34] Dharmapurikar S, Krishnamurthy P, Sproull T and Lockwood J, "Deep packet inspection using parallel bloom filters," IEEE Micro, Vol. 24, Issue 1: 52-61, Jan 2004

[35] Dharmapurikar S, and Lockwood J, "Fast and scalable pattern matching for network intrusion detection systems," IEEE Journal on Selected Areas in Communications, 24(10): 1781-1792, 2006

[36] DRC Coprocessor Information, http://www.drccomputer.com

[37] Durbin R, Eddy S, Krogh A, and Mitchison G, "Biological Sequence analysis: probabilistic models of proteins and nucleic acids", Cambridge University PRESS, ISBN: 9780521629713, 1998

[38] Faes P, Minnaert B, Christiaens, Bonnet E, Saeys Y, Stroobandt D, and Peer YVD, "Scalable hardware accelerator for comparing DNA and protein sequences," Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale), 33, 2006

[39] Felsenstein J, "Evolutionary trees from DNA sequences: a maximum likelihood approach," Journal of Molecular Evolution, 17(6): 368-376, 1981

[40] Fernandez E, Najjar W, and Lonardi S, "String matching in hardware using the FM-index," IEEE 19th annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 218-225, 2011

[41] Ferragina P, and Manzini G, "Opportunistic data structures with applications," Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS), 390, 2000

[42] Friedman N, Linial M., Nachman I, and Pe'er D, "Using Bayesian networks to analyze expression data," Journal of Computational Biology, 7: 601-620, 2000

[43] Garnier J, Osguthorpe DJ, and Robson B, "Analysis and implications of simple methods for predicting the secondary structure of globular proteins," Journal of Molecular Biology, 120:97-120, 1978

[44] Gotoh O, "An improved algorithm for matching biological sequences," Journal of Molecular Biology, 162(3): 705-708, 1982

[45] Gschwind M, Hofstee HP, Flachs B, Hopkin M, Watanabe Y, and Yamazaki T, "Synergistic processing in Cell's Multicore architecture," IEEE Micro, 26(2):10-24, 2006

[46] Hamming RW, "Error detecting and error correcting codes," Bell System Technical Journal, 29(2): 147-160, 1950

[47] Harris BB, "Acceleration of gapped alignment in BLASTP using the Mercury system," Technical Report, Department of Computer Science & Engineering, Washington University in St. Louis, 2006

[48] Henikoff S, and Henikoff JG, "Amino acid substitution matrices from protein blocks," Proceedings of the National Academy of Sciences of the United States of America(PNAS), 89(22): 10915-10919, 1992

[49] Herbordt MC, VanCout T, GU Y, Model J, and Sukhwani B, "Single pass, BLAST-like, approximate string matching on FPGAs," Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 217-226, 2006

[50] Ho J and Lemieux GGF, "PERG: a scalable FPGA-based pattern matching engine with consolidated bloomier filters," International conference on Field-programmable Technology (FPT), 73-80, 2008

[51] Homer N, Merriman B, and Nelson SF, "BFast: An alignment tool for large scale genome resequencing," PLoS ONE, 4(1): e7767, 2009

[52] Hussain HM, Benkrid K, Seker H, and Erdogan AT, "FPGA implementation of k-means algorithm for bioinformatics application: an accelerated approach to clustering Mircroarry data," NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 248-255, 2011

[53] Illumina sequencing-by-synthesis (SBS) technology, http://www.illumina.com/technology/solexa_technology.ilmn, 2011

[54] Jacob AC, "Design and analysis of an accelerated seed generation stage for BLASTP on the Mercury system," Master Thesis, Washington University in St. Louis, 2006

[55] Jacob AC, Lancaster J, Buhler J. and Chamberlain RD, "Preliminary results in accelerating profile HMM search on FPGAs," Proceedings of 6th IEEE international workshop on High Performance Computational Biology (HiCOMB), 1-8, 2007

[56] Jain A, Gambhir P, Jindal P, Balakrishnan M, and Paul K, "FPGA accelerator for protein structure prediction algorithm," 5th Southern Conference on Programmable Logic (SPL), 123-128, 2009

[57] Kent WJ, "BLAT: the BLAST-like alignment search tool," Genome Research, 12(4): 656-664, 2002

[58] Knodel O, Preusser T, and Spallek T, "Next-generation massive parallel short-read mapping on FPGAs," IEEE International Conference on Application-Specific Systems, Architecture and Processors (ASAP), 195-201, 2011

[59] Knowles G, and Gardner-Stephen P, "DASH: localizing dynamic programming for order of magnitude faster accurate sequence alignment," Proceedings of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB) , 732-735, 2004

[60] Krishnamurthy P, Buhler J, Chanberlain R, Franklin M, Gyang K, Jacob A, and Lancaster J, "Biosequence similarity search on the Mercury system," Journal of VLSI Signal Processing, 49(1): 101-121, 2007

[61] Krogh A, Brown M, Mian IS, Sjolander K, and Haussler D, "Hidden Markov Models in computational biology: applications to protein modeling," Journal of Molecular Biology, 235:1501-1531, 1994

[62] Kuon I, Tessier R and Rose J, "FPGA architecture: survey and challenges," Foundations and Trends in Electronic Design Automation, Vol 2, Issue 2, 135-253, 2008

[63] Lam TW, Sung WK, Tam SL, Wong CK, and Yiu SM, "Compressed indexing and local alignment of DNA," Bioinformatics, 24(6): 791-797, 2008

[64] Lancaster JM, "Design and evaluation of a BLAST ungapped extension accelerator," Master Thesis, Washington University in St. Louis, 2006

[65] Langead B, Trapnell C, Pop M, and Salzberg S, "Ultrafast and memory efficient alignment of short DNA sequences to the human genome," Genome Biology, 10(3), R25, 2009

[66] Lavenier D, Georges G, Liu X, "A reconfigurable index FLASH memory tailored to seed-based genomic sequence comparison algorithms," Journal of VLSI Signal Processing Systems, Vol. 48, Issue 3: 255-269, 2007

[67] Li H and Durbin R, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," Genome Research, 18:1851-1858, 2008

[68] Li H and Durbin R, "Fast and accurate short read alignment with Burrows-Wheeler transform," Bioinformatics, 25(14): 1754-1760, 2009

[69] Li H and Durbin R, "Fast and accurate long-read alignment with Burrows-Wheeler transform," Bioinformatics, 26(5): 589-595, 2010

[70] Li H, Handsaker B, Wysoker A, Fenell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R, 1000 Genome Project Data Processing Subgroup, "The sequence alignment/map format and SAMtools," Bioinformatics, 25(16): 2078-2079, 2009

[71] Li ITS, Shum W, and Truong K, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," BMC Bioinformatics, 8:185, 2007

[72] Li M, Ma B, kisman D, and Tromp J, "PatternHunter II: highly sensitive and fast homology search," Journal of Bioinformatics and Computational Biology, 2(3): 417-439, 2004

[73] Li R, Li Y, Kristiansen K, Wang J, "SOAP: short oligonucleotide alignment program," Bioinformatics, 24:713-714, 2008

[74] Li R, Yu C, Li Y, Lam TW, Yiu SM, Kristiansen K, and Wang J, "SOAP2: an improved ultrafast tool for short read alignment," Bioinformatics, 25(15):1966-1967, 2009

[75] Li R, Zhu H, Ruan J, Qian W, Fang X, Shi Z, Li Y, Li S, Shan G, Kristiansen K, Li S, Yang H, Wang J, and Wang J, "De novo assembly of human genomes with massively parallel short read sequencing," Genome Research, 20(2): 265-272, 2010

[76] Li Y, Terrell A, and Patel JM, "WHAM: a high-throughput sequence alignment method," Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD), 445-456, 2011

[77] Lin H, Zhang Z, Zhang MQ, Ma B, and Li M, "ZOOM! Zillions of oligos mapped," Bioinformatics, 24(21): 2431-2437, 2008

[78] Liu CM, Wang T, Wu E, Luo R, Yiu SM, Li Y, Wang B, Yu C, Chu X, Zhao K, Li R, and Lam TW, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," Bioinformatics , 28(6):878-879, 2012

[79] Liu Y, Maskell DL, and Schmidt B, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," BMC Research Notes, 2: 73, 2009

[80] Liu Y, Schmidt B, and Maskell DL, "Parallelized short read assembly of large genomes using de Bruijn graphs," BMC Bioinformatics, 12:354, 2011

[81] Liu Y, Schmidt B, and Maskell DL, "CUSHAW: a compatible short read aligner to large genomes based on the Burrows-Wheeler transform," Bioinformatics, 28(14):1830-1837, 2012

[82] Lloyd SP, "Least square quantization in PCM," IEEE Transactions on Information Theory, 28(2):129-137, 1982
[83] Lu M, Luo Q, Wang BQ, Wu JK, Zhao JX, "GPU-accelerated bidirected De Bruijn graph construction for genome assembly," Web Technologies and Applications LNCS, Vol. 7808: 51-62, 2013
[84] Ma B, Tromp J, and Li M, "PatternHunter: faster and more sensitive homology search," Bioinformatics, 18(3): 440-445, 2002
[85] Maizel JV, and Lenk RP, "Enhanced graphic matrix analysis of nucleic acid and protein sequences," Proceedings of the National Academy of Sciences of the United States of America(PNAS), 78(12): 7665-7669, 1981
[86] Mardis ER, "Next-generation DNA sequencing methods," Annual review of genomics and human genetics, 9:387-402, 2008
[87] Moraru I, and Andersen DG, "Exact pattern matching with feed-forward Bloom filter," Journal of Experimental Algorithmics (JEA), 17(1): 3.4, 2012
[88] Muriki K, Underwood KD, and Sass R, "RC-BLAST: towards a portable, cost-effective open source hardware implementation," Proceedings of the 19th IEEE international Conference on Parallel and Distributed Processing Symposium (IPDPS), 8:196.2, 2005
[89] National Instruments, "Introduction to FPGA technology: top 5 benefits," white paper, http://www.ni.com/white-paper/6984/en
[90] Navarro G, "A hybrid indexing method for approximate string matching," Journal of Discrete algorithms, 1(1): 205-239, 2000
[91] NCBI, http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html
[92] NCBI BLAST, ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast
[93] Needleman SB, and Wunsch CD, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," Journal of Molecular Biology, 48(3): 443-453, 1970
[94] Nickolls J, Buck I, Garland M, and Skadron K, "Scalable parallel programming with CUDA," ACM Queue, 6(2): 40-53, 2008
[95] Ning Z, Cox AJ, and Mullikin JC, "SSAHA: a fast search method for large DNA database," Genome Research, 11(10): 1725-1729, 2001
[96] Nourani M, and Katta P, "Bloom filter accelerator for string matching," Proceedings of 16th International Conference on Computer Communications and Networks (ICCCN), 185-190, 2007
[97] Oliver T, Schmidt B, and Maskell DL, "Reconfigurable architecture for bio-sequence database scanning on FPGAs," IEEE Transactions on Circuits and Systems II, 52(12): 851-855, 2005
[98] Oliver T, Schmidt B, Maskell DL, Nathan D, and Clemens R, "High-speed multiple sequence alignment on a reconfigurable platform," International Journal of Bioinformatics Research and Applications, 2(4): 394-406, 2006

[99] Olson CB, Kim M, Clauson C, Kogon B, Ebeling C, Hauck S, and Ruzzo WL, "Hardware acceleration of short read mapping," IEEE 20th annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 161-168, 2012

[100] Olsen GJ, Matsuda H, Haqstrom R, and Overbeek R, "fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood," Computer Applications in the Bioscience, 10(1): 41-48, 1994

[101] Pagh R, and Rodler FF, "Cuckoo hashing," Journal of Algorithms, 51: 122-144, 2004

[102] Pearson WR, "Rapid and sensitive sequence comparison with FASTP and FASTA," Methods in Enzymology, 183: 63-98, 1990

[103] Perrone M, and Petrini F, "Cell multiprocessor communication network: built for speed," IEEE Micro, 26(3): 10-23, 2006

[104] Pevzner PA, Tang H, Waterman MS, "An Eulerian path approach to DNA fragment assembly," Proceedings of the National Academy of Sciences of the United States of America(PNAS), 98(17):9748-9753, 2001

[105] Pournara I, Bouganis CS, and Constantinides GA, "FPGA-accelerated Bayesian learning for reconstruction of gene regulartory networks," International Conference on Field Programmable Logic and Applications (FPL), 323-328, 2005

[106] Ramakrishna MV, Fu E, and Bahcekapili E, "Efficient hardware hashing functions for high performance computers," IEEE Transactions on Computers, 46(12): 1378-1381, 1997

[107] Ramakrishna MV and Zobel J, "Performance in practice of string hashing functions," International conference on Database systems for advanced applications (DASFAA), World Scientific Press: 215-224, 1997

[108] Rizk G, and Lavenier D, "GASSST: Global alignment short sequence search tool," Bioinformatics, 26(20):2534-2540, 2010

[109] Ronquist F, Huelsenbeck JP, "MrBayes 3: Bayesian phylogenetic inference under mixed models," Bioinformatics, 19(12):1572-1574, 2003

[110] Rumble S, Lacroute P, Dalca A, Fiume M, Sidow A, and Brudo M, "SHRiMP: accurate mapping of short color-space reads," PLoS Computational Biology, 5(5): e1000386, 2009

[111] Saitou N, and Nei M, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," Molecular Biology and Evolution, 4(4): 406-425, 1987

[112] SciEngines, "Accelerated Smith-Waterman on RIVYERA hardware," white paper, http://sciengines.com/images/whitepapers/whitepaper_smith-waterman_on_rivyera_s3-5000.pdf

[113] Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, and Birol I, "ABySS: a parallel assembler for short read sequence data," Genome Research, 19(6): 1117-1123, 2009
[114] Smith TF, and Waterman MS, "Identification of common molecular subsequences," Journal of Molecular Biology, 147(1):195-197, 1981
[115] Sneath PHA, and Sokal RP, "Numerical taxonomy," Freeman, San Francisco, United States, 1973
[116] Sotiriades E, Kozanitis C, and Dollas A, "FPGA based architecture for DNA sequence comparison and database search," Proceedings of the 20th International Conference on Parallel and Distributed Processing Symposium (IPDPS), 193, 2006
[117] Staden R, "A strategy of DNA sequencing employing computer programs," Nucleic Acids Research, 6(7): 2601-2610, 1979
[118] Stamatakis A, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," Bioinformatics, 22(21): 2688-2690, 2006
[119] Stamatakis A, Ludwig T, and Meier H, "RAxML-III: a fast program for maximum likelihood-based inference of large phylogenetic trees," Bioinformatics, 21(4): 456-463, 2005
[120] Sternberg MJE, "Protein structure prediction: a practical approach," Oxford University Press, USA, 1997
[121] Tang W, Wang W, Duan B, Zhang C, Tan G, Zhang P, and Sun N, "Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator," IEEE 20th annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 184-187, 2012
[122] Tesla C2050 GPU computing processor, http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf
[123] TimeLogic, "Accelerated BLAST performance with Tera-BLAST: a comparison of FPGA versus GPU and CPU BLAST implementations," Technical Note, http://www.timelogic.com/documents/TimeLogic_Tera-BLAST_whitepaper_v1.0.pdf
[124] Ukkonen E, "On-line construction of suffix trees," Algorithmica, 14(3): 249-260, 1995
[125] Verilog tutorial, http://www.asic-world.com/verilog/veritut.html
[126] VHDL tutorial, http:// www.vhdl-online.de/tutorial
[127] Voelkerding KV, Dames SA, and Durtschi JD, "Next-generation sequencing: from basic research to diagnostics," Clinical Chemistry, 55(4): 41-47, 2009

[128] Vouzis PD, and Sahinidis NV, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," Bioinformatics, 27(2): 182-188, 2011

[129] Wang L. and Jiang T, "On the complexity of multiple sequence alignment," Journal of Computational Biology, 1(4): 337-348, 1994

[130] Weintraub B, "Building BLAST for coprocessor accelerators using Macah," Bachelor Thesis, Computer Science & Engineering, University of Washington, 2008

[131] WU BLAST, hg.wustl.edu/info/README.html

[132] Xia F, Dou Y, and Xu J, "FPGA-based accelerators for BLAST families with multi-seeds detection and parallel extension," The 2nd International Conference on Bioinformatics and Biomedical Engineering (ICBBE), 56-62, 2008

[133] Xia F, Dou Y, Lei G, and Tan Y, "FPGA accelerator for protein secondary structure prediction based on the GOR algorithm," BMC Bioinformatics, 12:S5, 2011

[134] Yu CW, Kwong KH, Lee KH, and Leong PHW, "A Smith-Waterman systolic cell," Proceedings of the 13th International Workshop on Field Programmable Logic and Applications (FPL), 375-384, 2003

[135] Zerbino DR, and Birney E, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," Genome Research, 18(5): 821-829, 2008

[136] Zhang Z, Schwartz S, Wagner L, and Miller W, "A greedy algorithm for aligning DNA sequences," Journal of Computational Biology, 7(1-2): 203-214, 2000

[137] Zierke S, and Bakos JD, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," BMC Bioinformatics, 11:184, 2010

List of Figures

List of Tables

List of Abbreviations

AOI	alignment of interest
ASIC	application specific integrated circuit
BFAST	BLAT-like fast accurate search tool
BFU	Bloomier filter unit
BLAST	Basic Local Alignment Search Tool
BLAT	BLAST-like alignment tool
BLOSUM	block substitution matrix
bp	base pair
BRAM	block RAM
BWT	Burrows-Wheeler transform
CAL	candidate alignment locations
CB	Computational Biology
Cell/BE	Cell Broadband Engine
CGC	Convey GraphConstructor
CLB	configurable routing blocks
CPU	central processing unit
CUDA	compute unified device architecture
DASH	Diagonal Aggregating Search Heuristic
DAWG	directed acyclic word graph
DCM	digital clock management
DMA	direct memory access
DP	dynamic programming
DP	data partitioning
EIB	Element Interconnect Bus
GOR	Garnier-Osguthorpe-Robson
GPU	graphic processing unit
FFBF	feed-forward Bloom filter
FPGA	field-programmable gate array

FPP	false positive probability
HDL	hardware description language
HMM	Hidden Markov Model
HPC	high performance computing
HPRC	high performance reconfigurable computer
HSP	high-scoring segment pair
IP	index partitioning
IOB	IO block
LIS	longest increasing subsequences
LPM	longest prefix matching
LUT	lookup table
MPHF	minimal perfect hash functions
MPI	message passing interface
MUMs	Maximum unique matches
NCD	native circuit description
NGD	native generic database
NGS	next generation sequencing
NIDS	network intrusion detection systems
NRE	none-recurring engineering
NW	Needleman-Wunsch
PAM	paint accepted mutation
PaPaRa	parsimony-based phylogeny-aware short read alignment
PAR	Place & Route
PBF	parallel Bloom filter
PCIE	Peripheral Component Interconnect Express
PE	processing element
PH	parallel hashing
PPBF	parallel partitioned Bloom filter
PPE	Power Processing Element
PRU	processing unit
QLT	quick lookup table

RISC	reduced instruction set computing
SA	suffix array
SAX	shift-add-xor
SCM	single cell module
SIMD	single instruction multiple data
SNP	single nucleotide polymorphism
SoPC	system-on-a-programmable-chip
SPE	Synergistic Processing Element
SPUs	score processing units
SSAHA	Sequence Search and Alignment by Hashing Algorithm
SSE2	Streaming SIMD Extensions 2
SW	Smith-Waterman
SWG	Smith-Waterman-Gotoh
TFBS	Transcription factor binding site
TNW	tiled-NW
TUC	Technical University of Crete
UCF	user constraint file
XPE	Xilinx Power Estimator