# Accelerating Genomic Sequence Alignment using High Performance Reconfigurable Computers

by

Peter Leonard McMahon

Submitted to the Department of Computer Science
in fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

University of Cape Town

October 2008

Supervisor: Dr Michelle Kuttel

Advisor: Prof. Yun Song, *Computer Science Division, University of California, Berkeley.*

## Abstract

Reconfigurable computing technology has progressed to a stage where it is now possible to achieve orders of magnitude performance and power efficiency gains over conventional computer architectures for a subset of high performance computing applications. In this thesis, we investigate the potential of reconfigurable computers to accelerate genomic sequence alignment specifically for genome sequencing applications.

We present a highly optimized implementation of a parallel sequence alignment algorithm for the Berkeley Emulation Engine (BEE2) reconfigurable computer, allowing a single BEE2 to align simultaneously hundreds of sequences. For each reconfigurable processor (FPGA), we demonstrate a 61X speedup versus a state-of-the-art implementation on a modern conventional CPU core, and a 56X improvement in performance-per-Watt. We also show that our implementation is highly scalable and we provide performance results from a cluster implementation using 32 FPGAs.

We conclude that reconfigurable computers provide an excellent platform on which to run sequence alignment, and that clusters of reconfigurable computers will be able to cope far more easily with the vast quantities of data produced by new ultra-high-throughput sequencers.

# Contents

# List of Figures

# List of Tables

*To my teachers.*

# Acknowledgements

The project described in this thesis has been a delight to work on, and I'm thrilled to have been involved in it. Foremost thanks goes to my project advisor at Berkeley, Prof. Yun Song, who was responsible for initiating and managing the bulk of the work that I present. This thesis happened by accident — it grew out of work I did for Yun part-time from September 2007 until April 2008. I initially aimed to "help out" with the development of a prototype design for sequence alignment, but I gradually became more involved, and my plan of spending a few days on the project turned into weeks, and then months. I had a great deal of fun learning about sequence alignment, and bioinformatics in general, from Yun during the course of my stay in Berkeley.

Dr Michelle Kuttel has been an excellent mentor to me for several years, so I was very pleased when the opportunity arose for me to work with her more formally. Michelle introduced me to parallel computing when I did a class project with her in 2004, and I've repeatedly used the knowledge I gained from that experience over the remainder of my university education. This thesis draws heavily on insights about parallel computing that I have learned from Michelle over the past few years. I'd also like to thank Michelle for being an excellent advocate for me in my dealings with the university bureaucracy, and for her very thorough reading of earlier drafts of this thesis.

I owe thanks to Prof. Michael Inggs in the Department of Electrical Engineering, Dr Alan Langman at the Karoo Array Telescope (KAT) project, and Dr Dan Werthimer at the Center for Astronomy Signal Processing and Electronics Research (CASPER) at the University of California, Berkeley for their support of me during this thesis. As I have mentioned, this thesis happened as an accident — the reason I was in Berkeley during 2007, and early 2008, was to work on radio astronomy instrumentation for KAT in the CASPER group, so without this official objective, I would never have had the opportunity to do my unofficial work on the sequence alignment project. I owe Dan special thanks for hosting me in Berkeley for 11 months, and for introducing me to much of the reconfigurable computing technology that I use in this thesis.

At Berkeley I worked closely with Henry Chen, Terry Filiba and Vinayak Nagpal on this project. Their contributions during our weekly meetings with Yun were invaluable. Henry deserves special mention for the many hours he spent with me helping me learn the Simulink toolflow. Kristian Stevens at the University of California, Davis, was responsible for the conception of the project, and the optimized alignment algorithm, and I'm glad to have worked with him.

Alex Krasnov, from the Berkeley Wireless Research Center[1], provided several useful ideas (including his suggestion of pipelining the design to obtain greater throughput), but I'm also indebted to him for his very generous help in setting up and teaching me how to operate his BEE2 cluster. Having worked next to Alex for several months, I appreciate how much time and effort (from both him and Dan Burke) went into creating the cluster.

During the course of this project, I solicited and received help from several researchers in the BEE2 community. I'm grateful to Fred Burghardt, Dan Burke, Chen Chang, Greg Gibeling, Jason Manley, Arash Parsa, Brian Richards and Hayden So for their assistance and advice.

Prof. Scott Hazelhurst from the University of the Witwatersrand provided several useful suggestions in his examiner's report, which I have incorporated into this thesis. I thank him for his very comprehensive assessment. Dr Craig Steffen from the University of Illinois at Urbana-Champaign, and Prof. Cathal Seighe from UCT also provided helpful comments on my writeup.

During my Masters I've had excellent administrative support from Regine Lord at UCT, Lee-Ann Bredeveldt and Niesa Burgher at KAT, and Stacey-Lee Harrison at the UCT Postgraduate Funding Office.

I thank my friends and family for their support.

The work in this thesis has been generously supported by the National Research Foundation in the form of a KAT Masters bursary, an NRF M.Sc. Scarce Skills Prestigious SET bursary, KAT travel funding, UCT Postgraduate Funding Office

[1]The BWRC is a research centre within the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley.

support, and U.S. National Science Foundation Grant No. 0619596 and Infrastructure Grant No. 0403427.

## Contributions

The work reported in this thesis had several contributors. In this section, I aim to describe specifically the contributions that I made, and mention who was responsible for the other work that enabled the developments detailed in this thesis.

Prof. Yun Song and Kristian Stevens were responsible for the project idea and the development of the optimized alignment algorithm. Henry Chen, Terry Filiba, Vinayak Nagpal and I met regularly with Prof. Song, and at these meetings Henry, Terry and Vinayak contributed many helpful ideas about overcoming limitations in the tools, and increasing the efficiency of the implementation.

I was responsible for the development of the sequence alignment algorithm implementation for the BEE2. I developed a scalable, lightweight streaming architecture, and verified the functionality of a parallel implementation of 10 simultaneous short-read alignments on a single FPGA. I built the DRAM controller/serializer that streams reference data through the processors. I developed the infrastructure to run alignments on several FPGAs simultaneously, and then to operate on several BEE2s simultaneously.

Henry was responsible for overcoming a limitation[2] in Xilinx System Generator to extend the number of simultaneous alignments to beyond[3] 10.

Terry wrote the optimized SSE CPU code against which the FPGA results are measured.

Alex Krasnov and Brian Richards showed us how to find the minimum of three

[2]The limitation was that Xilinx System Generator had an inefficient means of converting Simulink blocks to hardware description language, and as a result there was a practical upper limit on how many blocks in a diagram could be converted. We needed to exceed that limit.

[3]With a short-read length of 31 base-pairs, we were able to fit 18 alignment processors onto a single FPGA, thanks to Henry's work.

values in the most efficient way possible.

# Chapter 1

# Introduction

This thesis investigates the extent to which reconfigurable computing technology can be employed to address the computing challenges associated with new ultra-high-throughput genome sequencing technologies. Specifically, we investigate the performance of an optimized parallel sequence alignment algorithm implemented on a modern reconfigurable computer. In this introduction, we provide a brief background to and motivation for this investigation, provide details of the objectives, and outline the content of the thesis.

## 1.1 Background

One of the central goals of molecular genetics is the determination of the genetic basis of human disease. Considerable progress has been made in this regard in recent years; most notably the publication of the human genome in 2001 [1, 2], an effort that took two decades to complete. However, new genome sequencing technology is becoming available (for example, from Illumina [3]; see refs. [9, 10] for a technical discussion) that will dramatically reduce the amount of time it takes to produce sequence data from a sample of DNA. These advances will allow studies of human variation at the genetic level [4]. Such studies hold great promise for enabling important discoveries in determining which nucleotides and genes in the human genome hold information about human susceptibility to particular diseases.

There are considerable computing challenges [10] associated with the new sequencing products that arise due to the huge volumes of data that these machines produce.

The human genome consists of approximately 3 billion base-pairs[1]. The most expensive computational task in genome sequencing (using the dominant Whole Genome Shotgun Sequencing technique [7, 8]) is the alignment of short[2] fragments of the genome being sequenced against a reference genome. There may be hundreds of million of such fragments that need to be aligned in order to sequence a single genome. Current approaches to this problem using conventional computers may take tens to hundreds of thousands of CPU hours to complete the alignment requirements for the sequencing of a single human genome [1, 10, 11]. Even large cluster computers with thousands of CPUs may take several weeks to complete the necessary computations. Heuristic approaches, such as BLAST [6], reduce the computational burden, but result in a reduction in accuracy. Details of the dynamic programming-based algorithm that we used are contained in Section 2.3.3.

We have stated that genome sequencing is both an important application, and one that requires vast computational resources. The technology we use in this thesis to tackle this problem is *reconfigurable computing* [12]. A reconfigurable computer is a class of computer that includes a special-purpose processor that can perform specific tasks far more efficiently than a general-purpose processor can, but that is also reconfigurable. It is easy to see how a special-purpose processor can be designed to implement a particular algorithm far more efficiently than is possible using a von Neumann-architecture system. There exists a technology that enables us to build custom, special-purpose processors that can be reconfigured (i.e. reprogrammed) – "Field Programmable Gate Arrays" (FPGAs). FPGA processors provide the flexibility of software and the efficiency of Application Specific Integrated Circuits (ASICs)[3]. FPGAs were invented in 1984 [19], but, until recently, they have not had the required computational resources to compete with CPUs in the arena of high performance computing — their traditional use is to act as "glue logic" on electronic circuit boards, to connect I/O and components, rather than performing the bulk of the necessary computation (which is typically done by microprocessors or specialized digital sig-

[1]A "base-pair" can be thought of as a character from the alphabet $\Sigma = \{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$, and can hence be stored using $\log_2 |\Sigma| = \log_2 4 = 2$ bits.

[2]Short-read lengths from current sequencing technology range between 10 and 100 base-pairs.

[3]It is not actually accurate to claim that FPGAs give the same efficiency as ASICs, since FPGAs contain a large amount of interconnect to allow for reconfigurability that ASICs do not have. However, FPGAs do provide a comparable level of efficiency versus general-purpose CPUs to that which ASICs provide.

nal processing chips). Proofs-of-concept for FPGA-based reconfigurable computers existed as early as 1990 [45]. However, although projects such as SPLASH [45] and SPLASH 2 [46] successfully demonstrated the concept, the FPGA technology was not yet competitive with CPUs. The first commercially viable reconfigurable computing companies began circa 1996 [47], and more have appeared as FPGA technology has steadily improved.

Modern reconfigurable computers typically contain one or more FPGAs and a conventional CPU. The CPU is used to perform control tasks and operations that do not account for a significant percentage of the overall computation time. The CPU is typically used to run an operating system and associated services. Figure 1.1 shows a generic reconfigurable computer architecture. The CPU, FPGA and memory are all connected to a bus, over which the various devices can communicate. A common memory is a mechanism for allowing high-bandwidth communication between the CPU and FPGA. For example, the CPU might pre-process a portion of data and place it in the common memory. The FPGA can then process that data, and write the results back to the common memory, from where the CPU can obtain them.

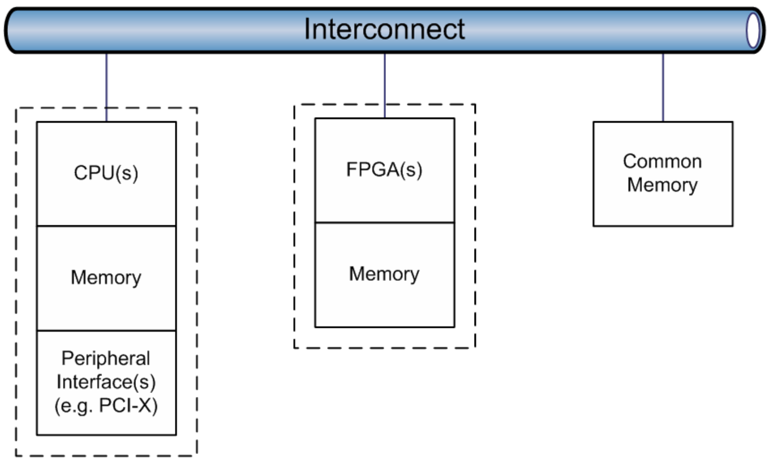

Figure 1.1: From [41]. A generic reconfigurable computer architecture.

The central challenge in porting applications to reconfigurable computers is the development of efficient FPGA designs that make the best use of the available resources. FPGA designs have traditionally been created using hardware description languages such as VHDL [48] and Verilog [49], which are well-suited to traditional digital design tasks, but are difficult languages in which to express scientific computing algorithms.

However, with the considerable increases in FPGA computational capacity that have occurred in recent years, and the subsequent commercial interest in reconfigurable computing, there has been a drive to produce tools for programming FPGAs that are more usable by developers who do not have experience with digital design.

The reconfigurable computing industry now claims successes in various applications areas, including financial modelling [31], bioinformatics [32] and oil and gas exploration (seismic data analysis) [33], amongst others. Of particular relevance to this thesis is the review by Hasan et al. [58] that summarizes recent achievements in Smith-Waterman sequence alignment on modern reconfigurable computers.

In this thesis, we build a sequence alignment implementation for the Berkeley Emulation Engine (BEE2) [14] reconfigurable computer using a tool chain that is based on the MATLAB Simulink graphical programming language [16, 17].

## 1.2 Objectives

The objective of this work was to investigate the performance benefits that can be gained in genomic sequence alignment for resequencing applications with reconfigurable computing technology. Specifically we aimed to:

1. Implement an optimized parallel sequence alignment algorithm on a BEE2 reconfigurable computer.
2. Extend this single FPGA alignment implementation to an implementation for a cluster reconfigurable computer.
3. Analyze the performance of the reconfigurable computer implementation over state-of-the-art software implementations.

The first requirement was to gain familiarity with the BEE2 platform and its associated development tools. Next, we aimed to implement a single sequence alignment custom processing engine. This would then need to be extended to a parallel alignment engine capable of aligning multiple sequences simultaneously. Once the single FPGA parallel alignment implementation was ready, this would be extended to a many-FPGA environment (e.g. a cluster of reconfigurable computers). Finally, we aimed to make an accurate appraisal of the performance of our implementation versus the most efficient alignment implementations for multi-core CPUs currently available.

## 1.3 Thesis Outline and Summary

This thesis is organized as follows:

Chapter 2 provides an introduction to reconfigurable computing, the BEE2 platform, and sequence alignment algorithms. We give an overview of acceleration techniques relevant to sequence alignment. We also present details of an optimized sequence alignment algorithm designed specifically for genome re-sequencing, which is the algorithm we implemented in this thesis.

Chapter 3 presents our implementation of the parallel sequence alignment algorithm on a single FPGA. This implementation is able to simultaneously align tens of short-reads against a reference genome. We give details of the operation of a single alignment processing engine, and of the efficient extension to multiple engines. We discuss our DRAM interfacing mechanism for streaming reference genome data from onboard memory, and our scheme for communication results to the control CPU[4]. We provide performance results, and make comparisons to equivalent software implementations. We also present the extension of our single-FPGA implementation to multiple FPGAs, and further to multiple BEE2 boards in a cluster.

Chapter 4 presents scaling, performance and power results for both our single FPGA and multiple FPGA implementations. We show that our architecture is scalable over a wide range of short-read lengths. We provide performance results from both our single FPGA implementation and from two highly optimized software implementations, and show that a single FPGA can achieve a 61X speedup over a single modern CPU core. We demonstrate that our cluster implementation scales linearly with the number of FPGAs, and show power consumption results that reveal a 56X improvement in performance-per-Watt in the FPGA implementation.

[4]The BEE2 runs Linux on an embedded PowerPC in the "Control FPGA". This CPU is used for control – loading programs, changing settings, and monitoring status.

Chapter 5 concludes this thesis, and provides some further analysis of the impact that reconfigurable computers can have on genome sequencing. Projected speedups for sequence alignment using current and near-future FPGA devices are also given.

The contributions of this thesis can be summarized as follows. To our knowledge, we have presented the first use of FPGAs for aligning, in parallel, multiple short sequences against a common long sequence using a dynamic programming-based algorithm. We have also presented the first implementation of and performance results from a new alignment algorithm that is optimized for short-read alignment in resequencing.

# Chapter 2

# Background

This thesis applies the technology and techniques of reconfigurable computing to the problem of sequence alignment. This chapter attempts to provide sufficient background in both areas such that bioinformatics experts can appreciate the reconfigurable computing implementation aspects, and those in the hardware acceleration community can follow the algorithmic aspects. Entire books have been written on both fields, so we can't hope give a comprehensive treatment of either in this chapter. We have provided several standard references that the interested reader can pursue. A comprehensive review of modern reconfigurable computing technology and techniques is available in [21], including several application case-studies.

In recent years, much attention has been given to acceleration of high performance computing applications using inherently parallel architectures. The creation of research centres such as Berkeley's Parallel Computing Laboratory [25] and Stanford's Pervasive Parallelism Laboratory [26] indicates that multicore processors will play a large role in the future. However, there has also been a resurgence of interest in other parallel computing techniques. Most prominent is the use of Graphics Processing Units (GPUs) to perform general-purpose computations [50]. Modern GPUs are effectively programmable many-core processors, albeit with many restrictions (particularly relating to memory hierarchy and arithmetic logic unit functionality) that are a legacy of their former exclusive use in graphics processing. IBM's multi-core Cell processor [51] is another entry into the field that provides yet another alternative parallel architecture for high performance computing applications.

Modern parallel computing technologies can generally be divided into two categories: replacement processors, and accelerators. Multi-core CPUs from Intel and AMD, and IBM's Cell processors, are intended as general-purpose CPUs that can run as standalone computing engines. Accelerators are designed to complement, rather than replace, existing CPU architectures. For example, GPUs are typically intended only to augment the processing capability of x86 or x64 microprocessor-based machines. In a common usage scenario with a CPU/GPU machine, the CPU stills runs the operating system and much of the application code, but the GPU is used where it can significantly outperform the CPU in operations that constitute a major fraction of the running time of a particular program. For example, if some application makes frequent calls to a Fast Fourier Transform routine, the FFT may be executed on the GPU, since the GPU can perform the FFT computation faster than the CPU can. Reconfigurable computers typically fall into the category of accelerator-based systems: a conventional CPU is paired with one or more FPGAs, which act as accelerators.

## 2.1 Reconfigurable Computing

The concept of a reconfigurable computer was initially proposed by Estrin [22, 23] in 1960. He proposed the development of a machine that contained a regular processor and an array of reconfigurable special-purpose processors. These special-purpose processors would provide excellent performance for the tasks they were tailored to do, but, importantly, could be easily reconfigured to implement different algorithms depending on the program that the computer was running.

It is easy to see how special-purpose hardware could provide more efficient computations than general-purpose processors. However, it was not until the invention of Field Programmable Gate Arrays in the 1980s that a technology existed with which it was possible to build special-purpose processors in hardware, and to be able to reconfigure the hardware for different tasks[1]. The development of the first gener-

[1]Application-Specific Integrated Circuits (ASICs) have long been used in single-application devices, such as cellular phones, digital cameras and suchlike, where performance and power efficiency are critical, the tasks it performs are known *a priori*, and there are sufficient volumes to justify the development of a custom chip. However, ASICs are not reprogrammable. FPGAs can, in some sense, be thought of as reprogrammable ASICs, and indeed the computer-aided design tools used

ation of reconfigurable computers occurred in the early 1990s, with projects such as SPLASH [45] and Splash 2 [46]. However, although these projects successfully demonstrated the concepts of reconfigurable computing, FPGA technology had not yet progressed to a point where reconfigurable computers were competitive with contemporary general-purpose microprocessors, let alone the supercomputers of the day.

### 2.1.1 Advantages of Reconfigurable Computers

In recent years, reconfigurable computing has acquired renewed popularity. This has been driven by many factors – the most important of which are the dramatic increases in the capabilities of modern FPGAs, and the fact that the clock frequencies of general-purpose microprocessors look unlikely to increase much beyond 3GHz. The latter is a result of the so-called "power wall", which provides a practical, physical limit to how much faster we can expect CPUs to run given cooling constraints [20]. This has led to the realization that computing faces a major challenge in moving wholesale to parallel computing, since almost all performance increases in the foreseeable future are likely to come from exploiting parallelism rather than relying on increased clock frequencies. The "Berkeley View" technical report [24] has been a popular call-to-arms to try to address these challenges. However, since FPGAs can be used to exploit parallelism at a much finer-grained level than is possible with multicore CPUs, reconfigurable computing has become an ever-more attractive proposition, or at least one that now warrants serious investigation to determine which problems can be more effectively solved on such machine architectures.

The computational density of FPGAs has improved over CPUs over the past several years, as shown in Figure 2.1. Now that CPU frequency increases can no longer be counted on, the metric of "performance per MHz" has increased in importance. FPGAs offer vastly superior theoretical integer arithmetic performance over CPUs. Of course, just as the "theoretical peak performance" of a CPU is not obtainable in typical applications, so too is it unusual for an application to be able to harnass the full computational power of an FPGA. Nevertheless, the orders-of-magnitude advantage that FPGAs have over CPUs in integer arithmetic density make them promising candidates for investigation of the performance it is possible to achieve in

by the ASIC and FPGA user communities are often very similar.

practice. To illustrate, the Xilinx Virtex 5 SX240T chip contains over 1000 multiply-and-accumulate units, each clockable at up to 550MHz [27]. This yields a peak performance of over 580 billion integer operations per second (580 GOPS). In comparison, the current fastest Intel CPU, the Intel Core 2 Extreme Quad-Core 3.2GHz chip, has a theoretical peak performance of 97 GOPS [28].

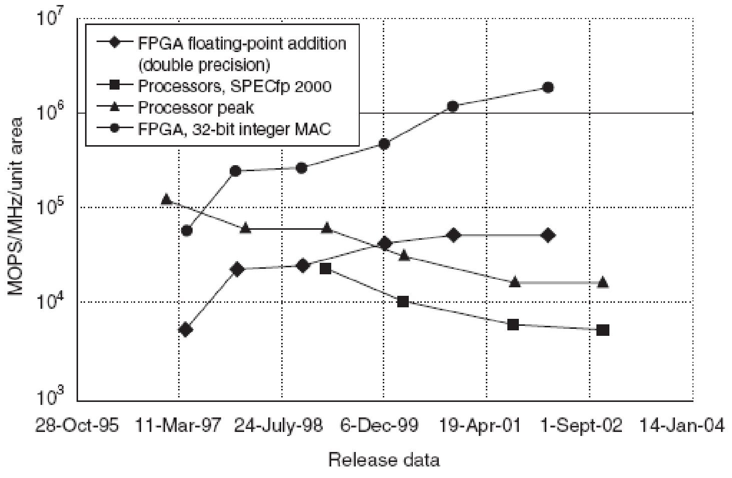

Figure 2.1: From [14]. FPGA and CPU Performance Scaling. Integer arithmetic on FPGAs now significantly outperforms CPUs per unit area and frequency.

However, theoretical peak performance is not the only metric of interest to high performance computer designers. Power performance is also an important consideration – in large HPC systems, power consumption and the associated cooling requirements are significant design factors. FPGAs are clocked at far lower frequencies than CPUs (an FPGA is typically clocked at less than 500MHz, whereas modern CPUs are clocked at between 2GHz and 3.5GHz). This leads directly to a significant difference in power consumption, since power consumption is proportional to the operating frequency [20]. Further, the performance of a system with a particular application is increasingly limited by the memory bandwidth available to the chip, rather than the theoretical peak arithmetic performance. This has lead to the so-called "memory wall" phenomenon [20], whereby performance is limited by the speed at which data can be inputted to and outputted from the processor. FPGAs have a benefit over CPUs in this area too, since modern FPGAs contain many I/O pins. The largest Virtex 5 has over 1700 pins, with 960 dedicated to high-speed serial I/O.

Major industry vendors such as Cray, SGI and HP now have products including FPGA accelerators, and many smaller hardware vendors have now also appeared on the market, such as SRC, Nallatech, DRC and XtremeData. Reconfigurable computing is certainly not an established technology and at present it seems unlikely that it will achieve broad adoption. However, there has been sufficient success in specific application areas to suggest that FPGAs will have some role to play as accelerators in high performance computing over the next decade. In the next section, we provide an overview of some competing approaches that may come to dominate the computer architecture landscale.

### 2.1.2 Measuring the Performance of Reconfigurable Computing Systems

We use the approach of Kindratenko et al. [38] to measure the performance of applications on reconfigurable computing systems. In many applications the "overhead" of pre- and post-processing data, and transferring data between an FPGA and a CPU, is not insignificant, and hence needs to be taken into account. When using FPGAs as accelerators, the total application time depends on several factors besides the time spent by the FPGA performing its computation. Specifically, there is typically pre-processing of data in the CPU before it can be sent to the FPGA. There is time to load the FPGA design, and to then transfer the data to the FPGA. Once the FPGA has performed its computation, the results need to be transferred back to the CPU, and there is often a post-processing stage required before the CPU can use the data. This is illustrated in Figure 2.2.

### 2.1.3 The Berkeley Emulation Engine 2

The BEE2 project [14] developed a reconfigurable computing board design that contains five FPGAs per board, where each FPGA includes two embedded PowerPC microprocessors. The BEE2 system has been used by many groups as a testbed for research on reconfigurable computing and has had significant successes in various application domains, including radio astronomy signal processing [34], systems biology [35] and processor emulation [55]. Programmability has long been a concern for reconfigurable computers and the approach taken with the BEE2 was to allow

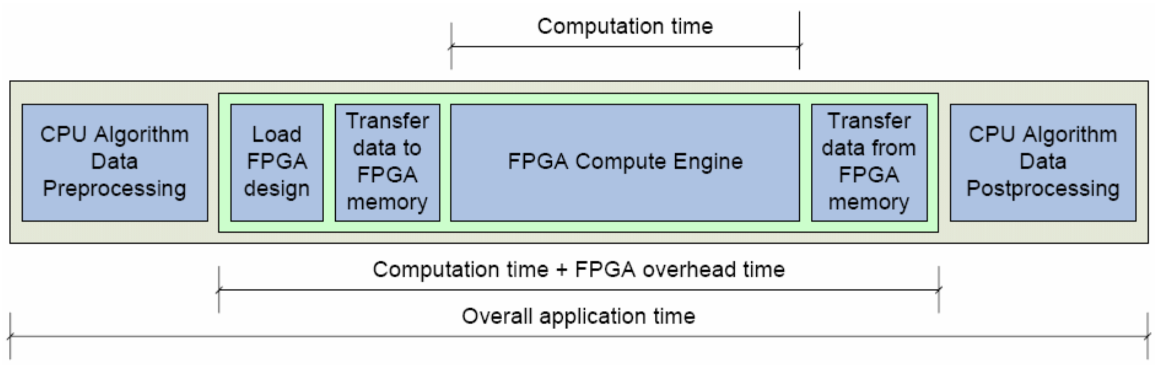
Computation time
CPU Algorithm Data Preprocessing
Load FPGA design
Transfer data to FPGA memory
FPGA Compute Engine
Transfer data from FPGA memory
CPU Algorithm Data Postprocessing
Computation time + FPGA overhead time
Overall application time

Figure 2.2: From [38]. Contributions to Total Computation Time in a Reconfigurable Computing System.

users to program the computer using a MATLAB Simulink-based graphical tool flow [16, 17, 18]. This works well compared to most other programming proposals, which typically rely on extensions (such as [13]) to fundamentally sequential languages to allow the programmer to express parallelism. The graphical tools also ease the learning curve for developers, over writing applications in hardware description languages such as VHDL and Verilog.

Figure 2.3 shows the high-level architecture of a BEE2 board. The four user FPGAs are each connected to the control FPGA using high-speed on-board links. There is also an on-board interconnect between adjacent user FPGAs. Each FPGA has four associated DRAM banks, which provides a significant amount of memory bandwidth to each chip. Also shown are the high-speed 10 gigabit Ethernet (10GbE) connections that are provided to allow the FPGAs to connect with other 10GbE devices, including other BEE2 boards. Figure 2.4 shows a photograph of a BEE2 board.

### 2.1.4 BORPH: A Reconfigurable Computing Operating System

BORPH [17, 18] is a Linux-based operating system designed to provide abstractions to users that allow reconfigurable computing platforms to be more easily used and programmed. BORPH's first target platform was the BEE2, and it has support for the BEE2's Simulink toolflow [16, 17, 18].

Figure 2.5 shows the high-level BORPH architecture. From a user's perspective,

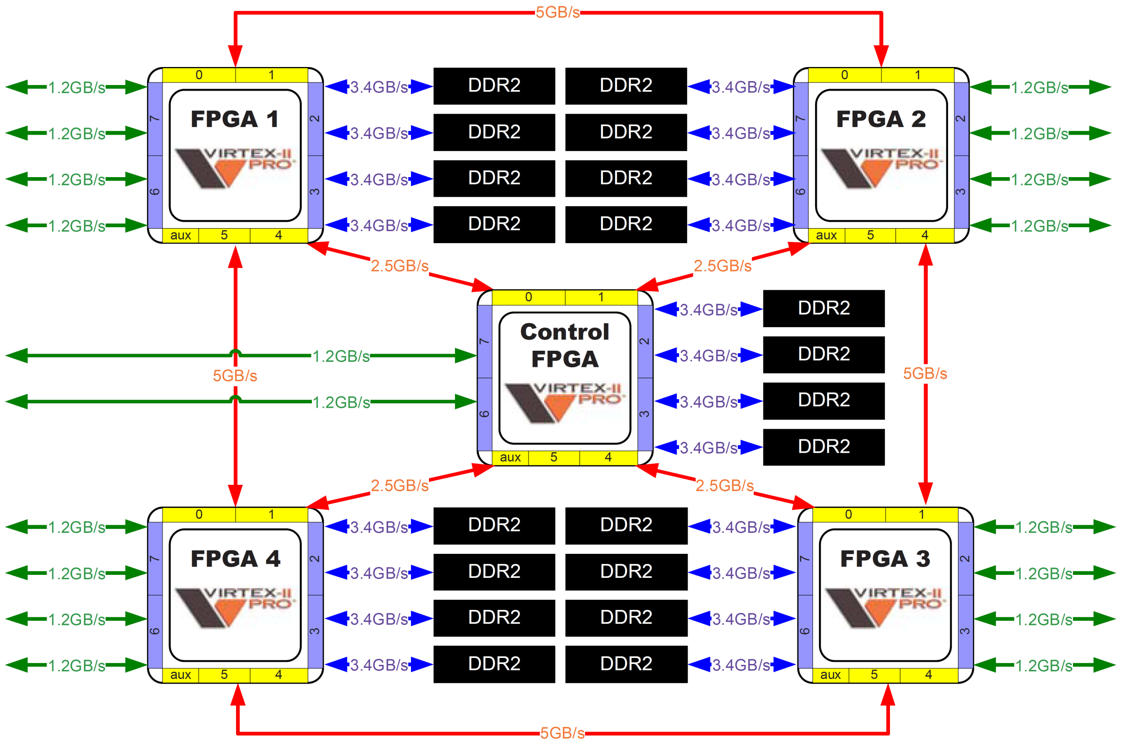

Figure 2.3: From [15]. BEE2 Architecture.

the important feature to notice in this diagram is that BORPH uses the concept of a “hardware process”. A hardware process refers to an FPGA design executing on an FPGA resource, which BORPH presents using a process abstraction. In BORPH, hardware processes behave in exactly the same way as software processes – the process appears in the process list, it is possible to redirect input and output to and from a process, it is possible to kill a process, and so on. BORPH manages the tasks associated with loading the FPGA design inside a “hardware executable” into a BEE2 user FPGA, and transferring data between the processor and the FPGA.

A further useful abstraction that BORPH provides is that of hardware resources in the FPGA. Specifically, it is possible to specify, at FPGA design time, that some registers and memories be “shared”. When that FPGA design is loaded with BORPH, the “shared” FPGA resources are made available to the user as files in the `/proc/` filesystem. It is therefore possible to interact with the FPGA design in BORPH purely by reading from and writing to files.

Figure 2.6 shows how BORPH operates on a BEE2. BORPH’s software primar-

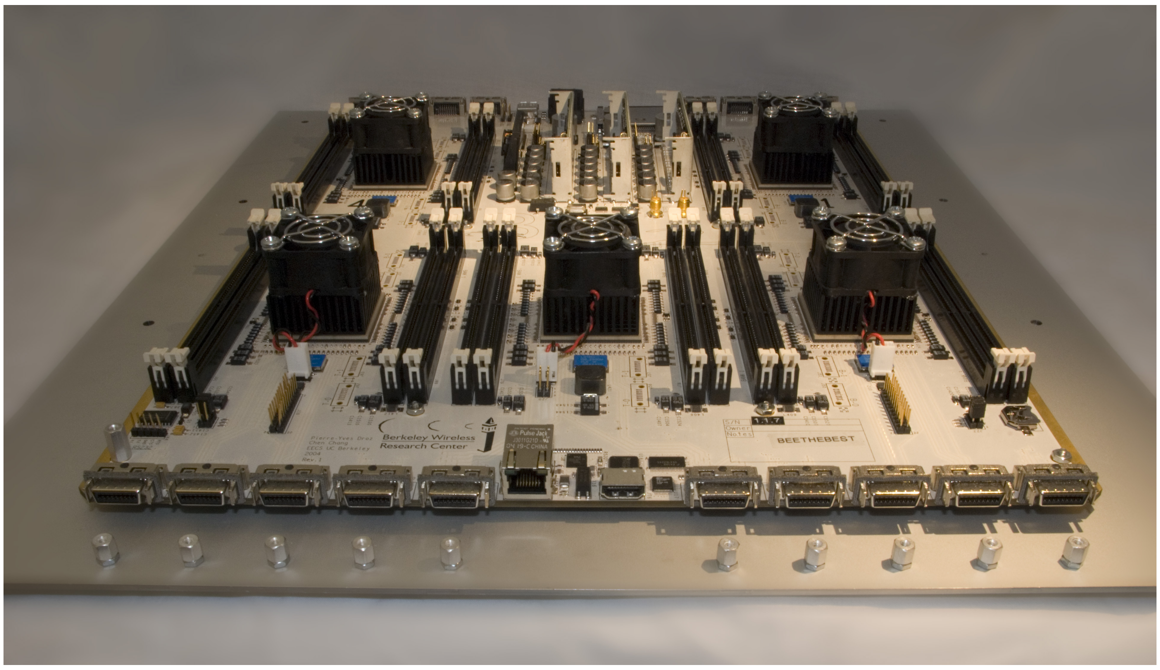

Figure 2.4: From [15]. BEE2 Board Photograph. The four user FPGAs and one control FPGA reside underneath the clearly visible black heatsinks. The DRAM slots (four per FPGA) and CX4 10GbE connectors (front of board) are also clearly visible.

ily executes on the PowerPC in the control FPGA. However, each user FPGA has a lightweight communications infrastructure that allows the control FPGA to communicate with it. BORPH uses the Xilinx SelectMAP bus [30] to reconfigure (reprogram) the user FPGAs as needed. An OPB bus within each FPGA is used to connect the "shared" resources in a particular design to the user FPGA PowerPC and hence ultimately to BORPH running on the control FPGA. BORPH performs all the necessary communications completely transparently.

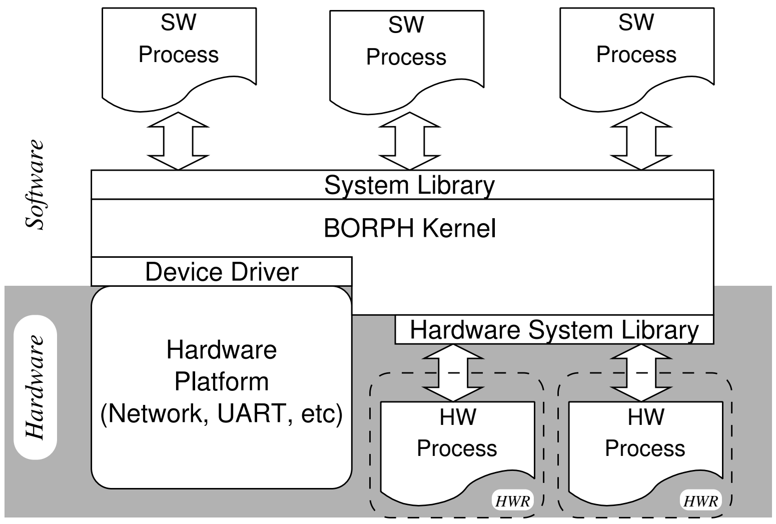

Figure 2.5: From [18]. BORPH Architecture.

To provide the support for shared resources in BORPH, it is necessary that the Simulink toolflow, which produces user FPGA designs, provide some interface with which BORPH can interact. Therefore, the toolflow has some knowledge of BORPH's requirements built in. Figure 2.7 shows a flow diagram of the tasks in the Simulink-based design environment. The user first describes his algorithm using the Simulink graphical design tool. Simulink provides a cycle-accurate simulator that lets the user fully debug his design before continuing. Once the design is ready, the toolflow can enter the "build" stage, in which it attempts to create an FPGA bitstream that implements the design described in Simulink. The first stage is the insertion of the interface logic that BORPH requires to communicate with resources that are marked in Simulink as being shared. The Xilinx System Generator product is then effectively used to generate hardware description language from the Simulink design. The Xilinx Embedded Development Kit is used to add support for the embedded PowerPC, and to appropriately connect the relevant buses. The Xilinx FPGA backend flow (ISE)

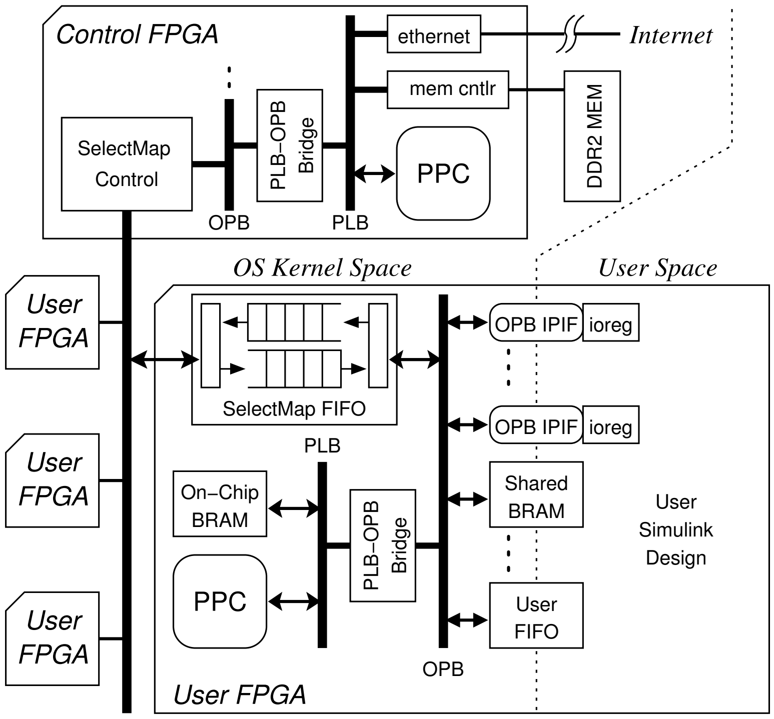

Figure 2.6: From [18]. BORPH Mapping to a BEE2.

creates a regular FPGA bitstream. Finally metadata about the design (such as the shared resources it contains) is added to the bitstream, and the result is a BORPH-executable file that will run as a hardware process.

## 2.2 Accelerating Applications by Exploiting Parallelism

Just as conventional computers are used in clusters by the high performance computing community, so too is it possible to build clusters of reconfigurable computers [55]. The same approaches to accelerating embarrasingly parallel programs that are used on CPU clusters [52] are equally applicable to reconfigurable computer clusters.

In order to outperform conventional CPUs, reconfigurable computers need to have FPGA designs that are efficient and that exploit parallelism. By "efficient", we mean that the number of clock cycles per algorithmic step should be as small as possible. The clock speed of a modern CPU is typically in the range 2–3GHz. The Virtex II Pro FPGAs on the BEE2 are clocked at 200MHz. Therefore reconfigurable com-

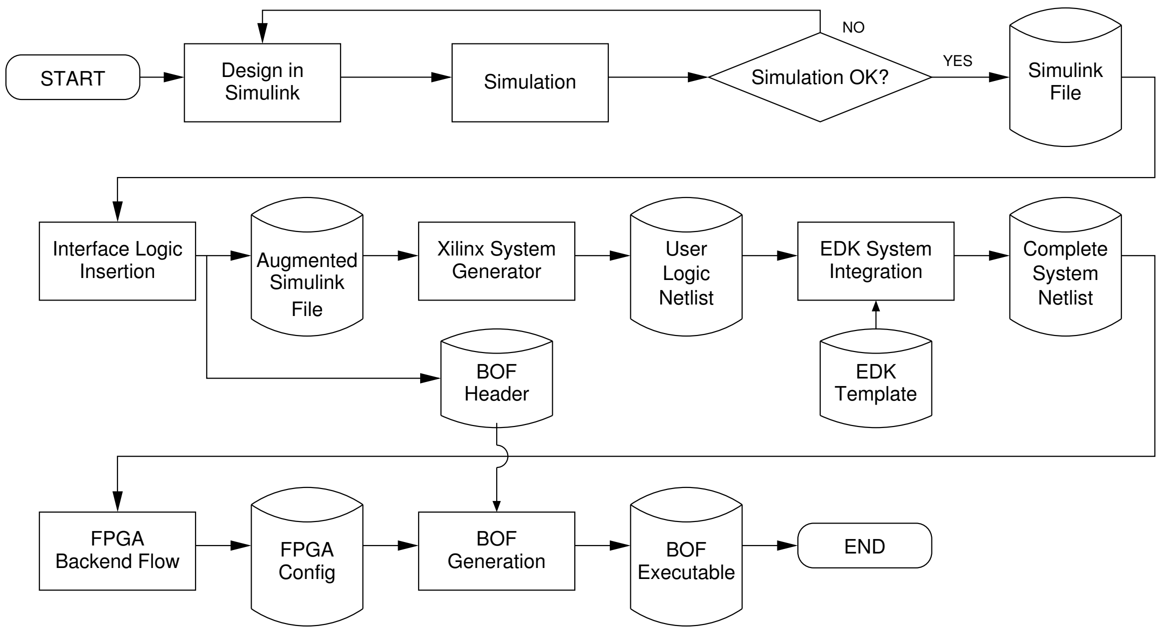

Figure 2.7: From [18]. Simulink Design Flow for BEE2 and BORPH.

puters have to overcome a 10X performance deficit just to reach parity with CPUs. Fortunately it is possible to produce FPGA designs that exploit the parallelism in the dynamic programming approach to sequence alignment, and to perform the requisite computations efficiently.

There is a considerable body of literature concerning parallel algorithms, and the mapping of applications to parallel architectures. Grama et al. [52] provide a comprehensive overview at an introductory level. In this thesis we are primarily concerned with two parallel computing concepts: *systolic arrays* and *embarrassingly parallel algorithms*.

The term "embarrassingly parallel" refers to computational problems where it is easy to decompose the computational load into self-contained subproblems that need little or no communication between them [29]; this is often called "wide" parallelism. Embarrassingly parallel problems are convenient problems to parallelize and they have excellent scalability – typically an embarrassingly parallel algorithm's performance will increase linearly with the number of processors used to execute it. This is in contrast to problems where communication between processors is necessary, which results in performance scaling being sublinear[2]. The problem of aligning many short-

[2]In the worst cases, beyond a certain number of processors performance cannot be improved, or

reads against a reference sequence is embarrasingly parallel in the sense that the alignment of each short-read is an independent computational problem.

Systolic arrays are a type of parallel architecture in which the processing task is divided amongst several (typically identical) processors in an array; systolic arrays exploit "deep" parallelism. Systolic arrays are characterized by the feature that the data flows synchronously across the array. In the one-dimensional case, data may flow from left-to-right, with each cell receiving input data from its left neighbour and outputting data to its right neighbour, as is shown in Figure 2.8. The cells operate independently of each other, with the exception of the data connections with their neighbours. Systolic arrays provide a convenient method for improving the time complexity of an algorithm at the expense of computational resources (FPGA fabric, in the case of reconfigurable computers). A big advantage of systolic arrays is that they are inherently scalable due to their use of only local connections between cells. In this thesis we use a systolic array design to implement a sequence alignment algorithm.

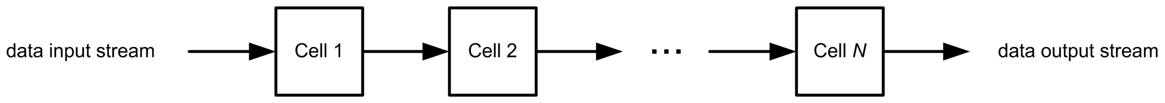

Figure 2.8: A Model Systolic Array.

## 2.3 Sequence Alignment

The general sequence alignment problem is a type of string matching and alignment problem [59]: given two strings, $r = r_1 r_2 \ldots r_N$ and $s = s_1 s_2 \ldots s_M$, we want to find the minimum number of (weighted) steps required to transform $s$ into $r$ (or, equivalently, $r$ into $s$). The possible edits to $s$ are: insertion of new characters, deletion of characters, and substitutions of characters. Typically, we are interested in assessing the similarity between two sequences; the number of edits is a measure of similarity. In this thesis, we are concerned with genomic sequences, which are strings over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$, i.e. $r_x \in \Sigma$ and $s_y \in \Sigma$ where $x = 1, \ldots, N$ and $y = 1, \ldots, M$. Sequence alignment is also routinely performed on protein sequences, which have a

there are severly diminishing returns as you add processors.

different, larger alphabet.

Edits to $s$ may be weighted, so as to prefer one type of edit (e.g. insertions) over another type (e.g. deletions). When assessing a proposed transformation of $s$ to $r$, we assign integer penalties to each edit: $\alpha$ for deletions in $s$, $\beta$ for insertions in $s$, and $\gamma$ for modifications of characters in $s$. $\alpha$ and $\beta$ are constants, but $\gamma$ may be a function of the character substitution (i.e. character mismatch). We can define the following matrix $\Gamma$ to specify mismatch penalties:

$$\Gamma = \begin{array}{c} \\ \texttt{A} \\ \texttt{G} \\ \texttt{C} \\ \texttt{T} \end{array} \begin{array}{c} \begin{array}{cccc} \texttt{A} & \texttt{G} & \texttt{C} & \texttt{T} \end{array} \\ \begin{pmatrix} \delta_{\texttt{AA}} & \delta_{\texttt{AG}} & \delta_{\texttt{AC}} & \delta_{\texttt{AT}} \\ \delta_{\texttt{GA}} & \delta_{\texttt{GG}} & \delta_{\texttt{GC}} & \delta_{\texttt{GT}} \\ \delta_{\texttt{CA}} & \delta_{\texttt{CG}} & \delta_{\texttt{CC}} & \delta_{\texttt{CT}} \\ \delta_{\texttt{TA}} & \delta_{\texttt{TG}} & \delta_{\texttt{TC}} & \delta_{\texttt{TT}} \end{pmatrix} \end{array}$$

The matrix elements $\delta_{\texttt{XY}}$ are integers, and correspond to the penalties of the character `X` transforming into character `Y`. If the penalty scheme is negative (i.e. $\alpha$ and $\beta$ are negative, and the best alignment is that which has the highest score), then typically $\delta_{\texttt{XY}} < 0$ when $\texttt{X} \neq \texttt{Y}$, and $\delta_{\texttt{XY}} \geq 0$ when $\texttt{X} = \texttt{Y}$. The latter case represents a character match, which should be rewarded.

To calculate the mismatch penalty for aligning two positions in the strings $s$ and $r$, you can perform a simple lookup in the penalty matrix $\Gamma$, using the character in $s$ to select the row, and the character in $r$ to select the column. Hence, if you wish to align the $i$th character in string $s$ against the $j$th character in string $r$, you can compute the penalty $\gamma(s_i, r_j)$ as $\gamma(s_i, r_j) = (\Gamma)_{f(s_i)f(r_j)}$ where $f : \Sigma \to \mathbb{N}$ is defined as the mapping from characters to (identical) row and column indices in $\Gamma$:

$$f(x) = \begin{cases} 0 & \text{if } x = \texttt{A} \\ 1 & \text{if } x = \texttt{G} \\ 2 & \text{if } x = \texttt{C} \\ 3 & \text{if } x = \texttt{T} \end{cases}$$

For example, if the character $s_i$ is `G` and the character $r_j$ is `T`, then the penalty is computed as $\gamma(s_i, r_j) = \gamma(\texttt{G}, \texttt{T}) = (\Gamma)_{f(\texttt{G})f(\texttt{T})} = (\Gamma)_{13} = \delta_{\texttt{GT}}$.

There are finitely many transformations of $s$ to $r$, but an exhausive search of all possible transformations results in an exponential time algorithm. Therefore to align long sequences, a different approach is needed.

### 2.3.1 The Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm [36] is an application of dynamic programming[3] to the *global alignment* problem. Global alignment is the problem we have just discussed: the transformation of a sequence $s$ to a sequence $r$.

To quantify the similarity between two sequences, we seek to compute a *similarity score* whose value is based on the optimal transformation of $s$ to $r$. For each required deletion and insertion of characters in $s$, we apply a penalty. Furthermore, we apply a penalty whenever the characters in $s$ and $r$ do not match. We denote the deletion penalty as an integer $\alpha$, the insertion penalty as an integer $\beta$, and the mismatch penalty an integer $\gamma(s_i, r_j)$ whose value is dependant on the specific mismatch that has occured.

As an example, following that in [60], let us set $\alpha = -2$, $\beta = -2$, and $\gamma(s_i, r_j)$ as:

$$\gamma(s_i, r_j) = \begin{cases} -1 & \text{if } s_i \neq r_j \text{ (mismatch)} \\ 1 & \text{if } s_i = r_j \text{ (match)} \end{cases}$$

When $\alpha = \beta$, we may refer to the insertion and deletion penalties simply as the *gap penalty* (which is the penalty for introducing a gap into either sequence, which can be interpreted as an insertion or deletion in $s$).

If we wish to align two sequences $s =$ `AAGCTCAGC` and $r =$ `ACGGCTAGC`, we can see that one potential alignment (with scores) is:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $s$ | A | A | G | $-$ | C | T | C | A | G | C |
| $r$ | A | C | G | G | C | T | $-$ | A | G | C |
| score: | $+1$ | $-1$ | $+1$ | $-2$ | $+1$ | $+1$ | $-2$ | $+1$ | $+1$ | $+1$ $= 2$ |

[3]Roughly, dynamic programming is a commonly used technique to solve problems that contain multiple overlapping subproblems. The solution can be computed in a reduced number of steps by keeping track of and using the solutions to already-computed subproblems.

The Needleman-Wunsch algorithm is used to compute the similarity score of the optimal alignment (and to output the alignment). The core part of the algorithm is the calculation of an $N+1$-by-$M+1$ *similarity matrix* $C$, often called the *dynamic programming table*. $(C)_{i,j}$ is the optimal similarity score for the alignment of the first $i$ characters of $r$ against the first $j$ characters of $s$. First set $(C)_{0,0}$ (the first cell in the matrix) to zero. Next initialize $(C)_{0,j} = \alpha \times j$ and $(C)_{i,0} = \beta \times i$. These represent the penalties for introducing gaps at the start of either sequence. We can complete the matrix using the following definition:

$$(C)_{i,j} = \max \begin{cases} (C)_{i,j-1} + \alpha \\ (C)_{i-1,j} + \beta \\ (C)_{i-1,j-1} + \gamma(s_i, r_j) \end{cases}$$

Again, following the example in [60], if we wish to align the sequences $s =$ `AGTC` and $r =$ `AGTAC`, using the same penalties as in the previous example, the completed similarity matrix is:

$$C = \begin{array}{c} \\ - \\ \mathtt{A} \\ \mathtt{G} \\ \mathtt{T} \\ \mathtt{C} \end{array} \begin{array}{c} \begin{array}{cccccc} - & \mathtt{A} & \mathtt{G} & \mathtt{T} & \mathtt{A} & \mathtt{C} \end{array} \\ \begin{pmatrix} 0 & -2 & -4 & -6 & -8 & -10 \\ -2 & +1 & 1 & -3 & -5 & -7 \\ -4 & -1 & +2 & 0 & -2 & -4 \\ -6 & -3 & 0 & +3 & +1 & -1 \\ -8 & -5 & -2 & +1 & +2 & +2 \end{pmatrix} \end{array}$$

The similarity score for the alignment of $s$ and $r$ in this case is $(D)_{N,M} = (D)_{5,4} = +2$.

From the similarity matrix it is possible to derive the alignment that resulted in the optimal score, provided that each element records which neighbour element was used to compute its value. However, here we are interested purely in the computation of similarity scores, and not in the reproduction of the alignments that produce them.

### 2.3.2 The Smith-Waterman Algorithm

The Smith-Waterman algorithm [37] is a modification of the Needleman-Wunsch algorithm that is used to compute *local alignments*. This is the most popular sequence alignment algorithm, since it allows for the alignment of parts of sequences. In global alignment, it is usually impossible to identify parts of sequences that are similar, since alignments that would identify such similarities will typically have been heavily penalized by misalignments of other parts of the sequences.

For example, if we have $s =$ `AAATAGTAGGTTGGAATTCCTATTT` and $r =$ `GGGGTTCCCAGTACCTAGG`, there are two substrings in each sequence that align perfectly (`AGTA` and `CCTA`). The Smith-Waterman algorithm can be used to easily reveal such substring alignments (local alignments), where global alignment algorithms will not, in general.

The modifications to the Needleman-Wunsch algorithm that result in the Smith-Waterman algorithm are as follows. First, the initialization of the similarity matrix $C$ is $(C)_{0,j} = 0$ and $(C)_{i,0} = 0$, i.e. each element in the first row and column is set to zero. This removes the penalty for placing gaps at the start of a sequence. Second, the definition of the remaining matrix elements is modified to be:

$$(C)_{i,j} = \max \begin{cases} (C)_{i,j-1} + \alpha \\ (C)_{i-1,j} + \beta \\ (C)_{i-1,j-1} + \gamma(s_i, r_j) \\ 0 \end{cases}$$

Hence no negative values may appear in the Smith-Waterman dynamic programming table. This allows parts of sequences that are similar to accrue positive scores within the matrix. The matrix will then contain several chains of increasing positive values in cases where there are subsequences that can be locally aligned. The final modification in Smith-Waterman results from the aforementioned fact — in Needleman-Wunsch the optimal similarity score is $(D)_{N,M}$, but in Smith-Waterman the optimal similarity score (for the best local alignment) is given by the element with the highest value (and is not necessarily $(D)_{N,M}$).

### 2.3.3 An Optimized Sequence Alignment Algorithm for Genome Resequencing Applications

In this thesis we are concerned with the speedup of sequence alignment for genome resequencing [61]. Resequencing is the task of sequencing DNA for an individual when provided with a reference sequence for the species. In resequencing, sequence alignment of many short-reads against a long reference sequence is the dominant consumer of computational resources. Current short-read sequencing technology produces short-reads of length approximately 30 base-pairs [3]. However, it is likely that new techniques will result in the short-read length doubling in the next few years. We assume that the reference sequence length, $N$, falls in the range $10^8 \leq N \leq 3 \times 10^9$, and the short-read length $M$ falls in the range $30 \leq M \leq 100$.

In genome resequencing we expect short-reads to represent contiguous regions of the genome being resequenced, but we do not know *a priori* from which location in the genome any given short-read is from. Therefore a suitable sequence alignment algorithm for resequencing is a hybrid of the Needleman-Wunsch and Smith-Waterman algorithms – we want to align the entire short-read, not parts of it, but we do not want to penalize gaps at the start or end of the short-read. The Needleman-Wunsch algorithm with the Smith-Waterman modification to initialize the first row and column in the similarity matrix to zero is suitable.

Stevens and Song [42] have suggested a sequence alignment algorithm based on the Needleman-Wunsch and Smith-Waterman algorithms that is sufficiently flexible for resequencing applications, but is also more efficiently implementable on FPGAs than the Needleman-Wunsch or Smith-Waterman algorithms. They argue that it is not necessary to allow for arbitrary integer penalties. They suggest that the penalties $\alpha$, $\beta$ and $\gamma$ be restricted to positive integers in the range $[0, 3]$. The restriction to positive values implies that the matrix definition should select a minimum, rather than maximum, value:

$$(C)_{i,j} = \min \begin{cases} (C)_{i,j-l} + \alpha \\ (C)_{i-1,j} + \beta \\ (C)_{i-1,j-1} + \gamma(s_i, r_j) \end{cases}$$

Furthermore, they suggest that each element in the matrix can be stored using a 6-bit value, provided that saturating logic[4] is used.

The final difference between conventional sequence alignment algorithm implementations and the one presented in this thesis is that we are only required to output the values from the final row in the matrix. In fact for this application, it is sufficient to output only the column positions where the value in the final row is below some threshold $T$.

In this thesis we implemented Stevens and Song's optimized alignment algorithm.

[4]In $b$-bit saturating logic, any register assignment of a value greater than $2^b - 1$ results in the value $2^b - 1$ being stored. The crucial point is that the register saturates rather than wraps.

# Chapter 3

# Multiple Sequence Alignment Implementation on a Cluster Reconfigurable Computer

In this chapter, we present the design and implementation of a custom processor design on a single FPGA that can simultaneously align tens of short-reads against a common reference genome. We implemented this design on a single user FPGA on a BEE2 reconfigurable computer. We then extended it from a single FPGA system to a multiple FPGA system that spans eight BEE2 processing boards. In the multiple FPGA system we replicated the single FPGA design across several FPGAs and implemented a control infrastructure to manage execution. Since the alignment algorithm is embarrassingly parallel, the primary concern in extending the system beyond a single FPGA was ensuring that the overhead from communicating input data to, and results from, the FPGAs does not significantly affect performance as we scale to larger numbers of FPGAs.

## 3.1 A Streaming Short-read Alignment Implementation

As discussed in Chapter 2, aligning multiple short-reads against a reference genome is an embarrassingly parallel problem – it is possible for each alignment to be completed independently of the others. Therefore, the first stage in the development of an FPGA design to simultaneously align multiple short-reads is the development of a design to align a single short-read. This design can be easily extended to support simultaneous multiple alignment.

### 3.1.1 Wavefront Parallelism in Dynamic Programming Sequence Alignment

The dynamic programming approach to sequence alignment, as pioneered by Needleman and Wunsch [36], and Smith and Waterman [37], allows for the update of many cells in the dynamic programming table in parallel, provided that appropriate sets of cells are chosen. This is called "wavefront parallelism" because the sets of independent cells appear as "waves". Consider a dynamic programming table[1], where $N$ columns represent the reference base-pairs, and $M$ rows represent the short-read base-pairs. Clearly, it is not possible to compute cell updates for entire rows or columns in parallel, since there are data dependencies between cells in such sets. However, if we choose our sets of cells to be those that are in anti-diagonals, there are no dependencies between cells in a set. This is illustrated in Figure 3.1, which highlights a single anti-diagonal set of cells. This technique was introduced by Lipton and Lopresti in 1985 [39] to build a custom processor for fast string comparisons, and is routinely used in parallelizations of dynamic programming sequence alignment algorithms for FPGAs (see Hoang's pioneering paper [40], and refs. [56, 57, 58] for more recent work).

We can see from the staggering in the dynamic programming anti-diagonal scheme that, if the cell on row $i$ for anti-diagonal $t$ needs the reference base-pair from column $j$, then for the next anti-diagonal, $t+1$, the cell on row $i+1$ will need the reference base-pair from column $j$. In addition, in order to update the cell at row $k$ and column $l$, $C(k,l)$, for anti-diagonal $t$, we also need the values from cells $(k-1,l-1)$, $(k-1,l)$ and $(k,l-1)$. These values were calculated during the updates for anti-diagonals $t-1$ and $t-2$.

This observation has a very important consequence: since the cell updates, when computed in anti-diagonals, only require data from the previous two anti-diagonal

[1]The table should have dimensions $N+1$-by-$M+1$, where the first row and column are each filled with zeros. We use a zero-based indexing notation.

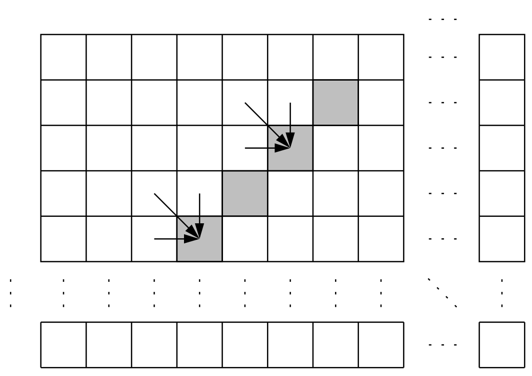

Figure 3.1: From [42]. “Wavefront Parallelism” in Dynamic Programming Sequence Alignment. If cells in the dynamic programming table are updated in anti-diagonals, each update can be calculated in parallel due to the arrangement of the data dependencies (shown as arrows for two example cells).

computations, it is only necessary to store the states of the cells in three anti-diagonals at any point in time. Moreover, the data dependencies exhibit a high degree of spatial and temporal locality (including the dependencies on the reference sequence base-pairs), which enables the implementation of the computation using a systolic array that only requires communication between adjacent elements in the array. With the parallel anti-diagonal approach, only approximately 3 anti-diagonals $\times M$ rows $= 3M$ values need to be simultaneously stored to carry out each short-read alignment. Since typically $M = 30$ (and we expect $M$ to have an upper bound of 100), the required storage will be 90 bytes[2] per short-read alignment, which can easily be supplied using registers (single-clock latency memory). A serial approach, which computes a single new cell value at each step, would require far less storage, but since low-latency storage for our application of dynamic programming alignment (where $M < 100$) is not a limiting constraint, we don't need to consider further any potential tradeoffs between space and time complexity.

The following listing is a (C-like) pseudocode description of the anti-diagonal technique[3], implemented for serial execution. The function `compute_scores` outputs the dynamic programming table values from the last row[4]. Three arrays, each of length

[2]This value is calculated using the assumption that each cell's value is stored using 8 bits. In practice we use a 6-bit register, so the memory required is actually $\approx 68$ bytes when $M = 30$.

[3]The pseudocode listing is based on a C implementation by Yun Song.

[4]Because the mismatch, deletion and insertion penalties are assumed to be positive, in practice

$M+1$, are created as storage for the current anti-diagonal (stored in `this`), the anti-diagonal from the previous update (stored in `last`), and the anti-diagonal from one further previous update (stored in `lastlast`). The arrays are appropriately initialized, and the remaining pseudocode is the main loop that computes the table values, one anti-diagonal at a time. The innermost `for` loop computes the value of each cell in a single anti-diagonal; this is the loop that can be parallelized because there are no conflicting data dependencies in this code.

```
function compute_scores(shortread, shortread_length, reference, reference_length)
 // declare arrays
 int this[shortread_length+1]
 int last[shortread_length+1]
 int lastlast[shortread_length+1]

 // initialize arrays
 this[0] = 0
 last[0] = 0
 lastlast[0] = 0
 for i = 1 to shortread_length
   this[i] = MAXINTEGER
   last[i] = MAXINTEGER
   lastlast[i] = MAXINTEGER
 // Perform DP table updates
 for i = 1 to reference_length+shortread_length-1
   // Move anti-diagonal arrays
   lastlast = last
   last = this

   // Update the current anti-diagonal
   for j = 1 to shortread_length
     // Insertion or deletion
     this[j] = min(last[j-1] + INS_PENALTY, last[j] + DEL_PENALTY)
     if (i-j < reference_length) and (i >= j) then
```

we will only be interested in the outputs where the dynamic programming table cell value was under some threshold, but we ignore this detail for now.

```
      if (shortread[j-1] != reference[i-j]) then // mismatch
        this[j] = min(this[j], lastlast[j-1] + MISMATCH_PENALTY)
      else // match
        this[j] = min(this[j], lastlast[j-1])
  print i-shortread_length+1 // output position in reference
  print this[shortread_length] // output last row value
```

A systolic array can be used to compute the cells of the dynamic programming table in parallel using the wavefront technique. A high-level diagram of such an array is shown in Figure 3.2. Each anti-diagonal has $M$ cells, and each cell can be updated in parallel, so the systolic array consists of $M$ "Processing Engines" (PEs) that each compute a new value for the cell in the row that they are responsible for. PE1 computes the values for row 1, PE2 computes the values for row 2, and so on.

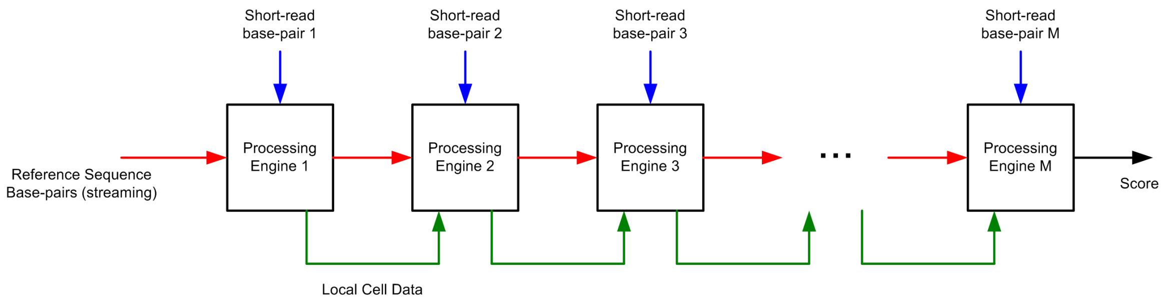

Figure 3.2: Systolic Array for Streaming Parallel Sequence Alignment. Each "Processing Engine" computes a single cell update. The reference sequence is streamed through the Processing Engines. Each PE passes local cell information to the following PE. The output from the final PE is the value for the cell in row $M$ in the dynamic programming table, where the column is that which corresponds to the reference sequence base-pair that has most recently reached the final PE.

The output of the systolic array implementation is very similar to that of the pseudocode implementation: the final PE's output is the cell value for the last row in the dynamic programming table. In the pseudocode there is an explicit output of the column (reference base-pair). In the systolic array implementation, it is necessary to keep track of how many clock cycles have elapsed since the start. From this number it is possible to deduce which column the current output is for.

Another subtlety in the systolic array implementation is that the PEs are assumed to continuously compute new values based on their inputs, so there is no explicit start and end to the "program" as there is in a traditional programming implementation. To define a start, we need to reset the registers in the PEs (this corresponds to the initialization steps in the pseudocode). It is then necessary to calculate how many clock cycles a particular alignment is expected to take (and this is deterministic, since the systolic array is not time-slice scheduled – it is always running, and runtime is directly proportional to the length of the reference, $N$), and to disable output after this many clock cycles have elapsed since beginning an alignment computation.

### 3.1.2 A Cell Update Processing Engine

In order to update a cell $C(k, l)$, the following information is necessary:

1. The values $C(k-1, l-1)$, $C(k-1, l)$ and $C(k, l-1)$.
2. The reference base-pair value for column $l$, $r_l$.
3. The short-read base-pair value for row $k$, $s_k$.

As we have discussed above, these data dependencies are local and allow the possibility of developing an efficient systolic array implementation of the algorithm, whereby values are "streamed" through an array of processing engines. Each processing engine is responsible for updating a single cell, and collectively the array of processing engines updates all the cells in a single anti-diagonal. Each processing engine, over the course of the algorithm's execution, computes the cell updates for a particular row.

For a single update, processing engine $k$ has to compute a value for cell $(k, l)$, for the column $l$ corresponding to antidiagonal $t$. The processing engine has as inputs for this computation the reference base-pair $r_l$, the short-read base-pair $s_k$, and cell values $(k-1, l-1)$, $(k-1, l)$ and $(k, l-1)$. The value of $C(k, l)$, the cell $(k, l)$, is computed as:

$$C(k, l) = \min \begin{cases} C(k, l-1) + \alpha \\ C(k-1, l) + \beta \\ C(k-1, l-1) + \gamma(s_k, r_l) \end{cases}$$

$\alpha$ is the penalty applied for a deletion in the short-read sequence (a character insertion in the reference sequence), $\beta$ is the penalty applied for an insertion in the short-read sequence, and $\gamma(s_k, r_l)$ is the penalty applied for a base-pair mismatch (i.e. substitution) between the reference sequence and short-read sequence. In general $\gamma(a, b)$ is defined piecewise for all combinations of $a, b \in \{A, C, G, T\}$. In our implementation,

$$\gamma(s_k, r_l) = \begin{cases} 0 & \text{if } s_k = r_l \\ \delta & \text{otherwise} \end{cases}$$

Therefore no penalty is applied if the base-pairs are the same, and a standard penalty of $\delta$ is applied if they are not.

The penalties are constrained to be integers in the range 0 to 3, and can therefore be represented using 2 bits.

The steps involved in computing a cell update are:

1. Compute $\gamma(s_k, r_l)$.
2. Compute $C(k-1, l-1) + \gamma(s_k, r_l)$.
3. Compute $C(k, l-l) + \alpha$
4. Compute $C(k-1, l) + \beta$
5. Compute minimum.

Steps 1, 3 and 4 can be performed in parallel. However, step 2 requires the result from step 1. Step 5 can only be performed once the result of step 2 is known. These dependencies hint that it isn't possible to create a high-performance design where the cell update is computed in a single clock cycle. Instead, we need to perform the computation in stages.

If we provide each processing engine with registers to store the values of the cells in the previous column, and one column before that (corresponding respectively to the `last` and `lastlast` variables in the pseudocode), and we require that each processing engine pass this data along to the next processing engine, then every PE will have access to the data it needs. Figure 3.3 shows how processing engine $k$ (corresponding to row $k$) requires, for its part in the computation of antidiagonal $t$, values computed in antidiagonals $t-1$ and $t-2$ by processing engine $k-1$, in addition to the value it computed for antidiagonal $t-1$.

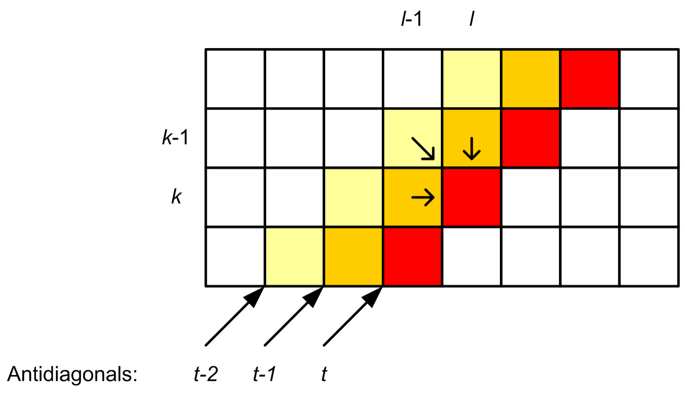

Figure 3.3: Data Dependencies in Antidiagonal Cell Update Computations. Processing Engine $k$ requires the `last` ($l$) and `lastlast` ($l-1$) values from Processing Engine $k-1$, and the `last` value from itself, in order to compute the update for cell $(k, l)$, which is part of antidiagonal $t$.

Figure 3.4 shows a high-level view of a single processing engine, emphasizing its state variables and inputs/outputs. Specifically each PE must store its current (to be computed) value (`this`), the previous value it computed (`last`), and the value before that (`lastlast`). At every new update, PE $k$ receives the `last` and `lastlast` values from the previous processing engine, PE $k-1$, and passes its own state variables on to PE $k+1$. In addition, PE $k$ passes the reference sequence base-pair it last worked with on to PE $k+1$, and receives its next reference base-pair to work with from PE $k-1$.

In the dynamic programming table for sequence alignment, we assign a short-read base-pair to each row. Each PE then has a user input that defines the short-read base-pair with which it computes mismatches with the reference base-pairs that are streamed through it. This short-read base-pair input remains constant throughout the execution of an alignment.

The PE has an "enable" input that needs to be connected to a clock source that has a frequency four times smaller than that of the FPGA. This is used for synchronization: only after four FPGA clock cycles have occurred should a PE write values to its registers.

Finally, each PE has a "reset" input. This allows the register values to be reset. Recall from the previous section that the systolic array continually computes values. Therefore, we need a means to initialize the values in each processing engine when we wish to start the computation of an alignment; providing an individual[5] reset to each PE allows us to do this.

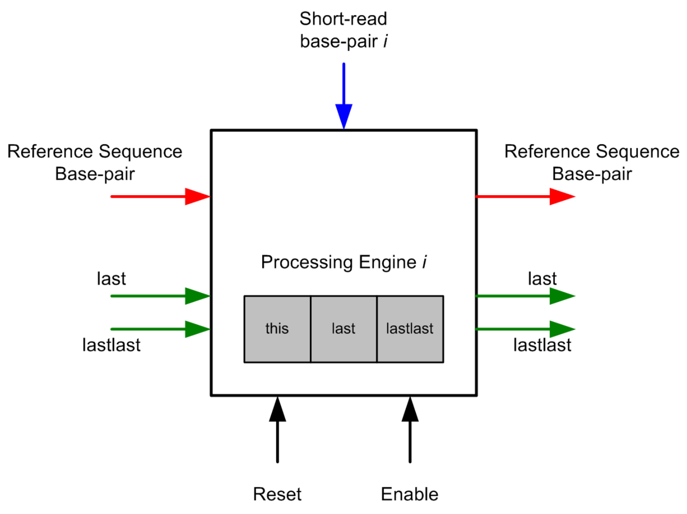

Figure 3.4: A High-Level View of a Cell Update Processing Engine. A single PE performs a single cell update for a particular row $i$ in an anti-diagonal. Its streaming inputs are: the next reference base-pair (corresponding to column $l$), and the cell values for its diagonal $(k-1, l-1)$ and upper $(k-1, l)$ dependencies (which both come from the row above it, hence the values can be streamed from the previous PE).

Figure 3.5 shows a partitioning of tasks in a processing engine. The tasks are divided into three stages, based on the data dependencies between them.

[5]We will discuss shortly why it is necessary to have an individual reset for each processing engine, rather than some global reset – although from the perspective of the PE itself, there is no difference between the two.

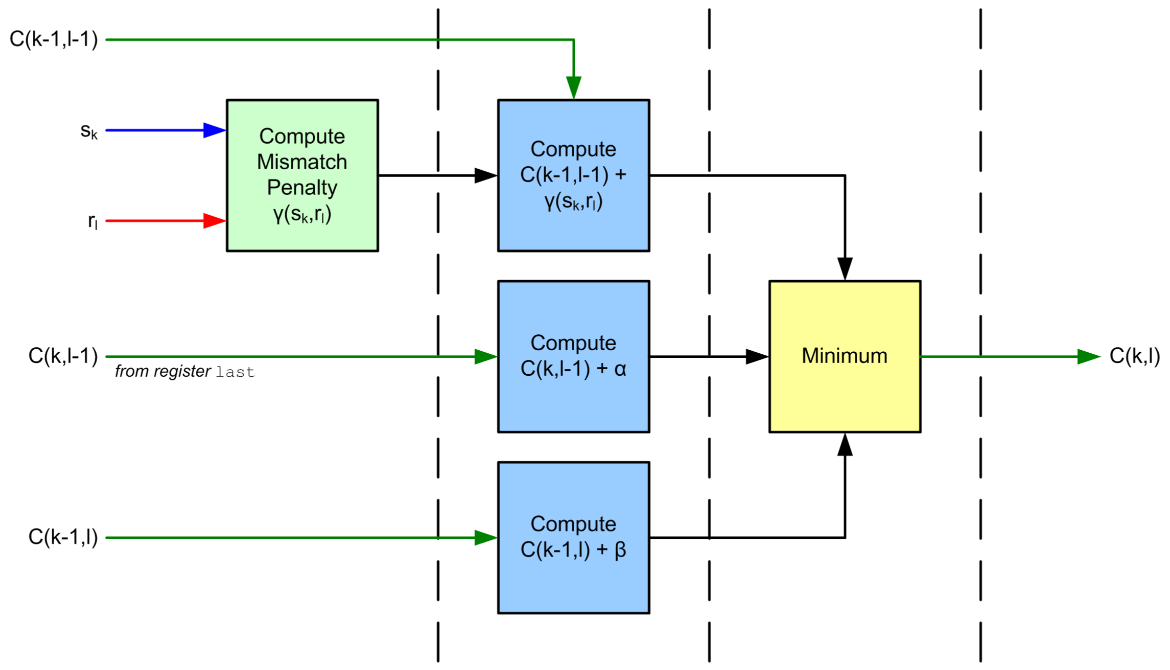

Figure 3.5: Computation Stages in a Processing Engine. All the inputs are external, except for $C(k, l-1)$, which is obtained from the register `last` (not shown) in this processing engine.

Since the FPGA resources used by the processing engine design is ultimately the the primary limiting factor of the performance of our implementation, we desired to make the PE as efficient as possible. This implies the need to optimize the computations performed in a single PE. Through trial-and-error[6] we managed to determine that the computation of $C(k-1, l-1) + \gamma(s_k, r_l)$ can be done in one clock cycle without reducing the clock frequency at which we could run the design[7]. The calculations of $C(k, l-l) + \alpha$ and $C(k-1, l) + \beta$ also require only one clock cycle. We used an optimized method of computing the minimum of three integers[8], which is composed of two instances of a "minimum of two integers" design. This enabled us to perform the computation of the minimum in two clock cycles. A fourth clock cycle was used for a "write back" stage to write the result to a register.

[6]To readers not familiar with the state of FPGA design tools, this might sound terribly unscientific and reflect badly on our methods, but unfortunately many questions about timing and resource usage can only be answered in this way.

[7]Without the use of an external clock source, the BEE2 can run at a clock frequency of either 100MHz or 200MHz – we aimed to make our designs meet timing for 200MHz.

[8]This subsystem design was kindly supplied to us by Alex Krasnov and Brian Richards.

Figure 3.6 shows the subsystem that computes the value $C(k-1, l-1) + \gamma(s_k, r_l)$. The comparator returns 0 if $s_k = r_l$, and 1 otherwise. In parallel with the comparator is an adder that computes the sum $C(k-1, l-1) + \delta$. Both this value and the unpenalized score $C(k-1, l-1)$ are supplied as inputs to a multiplexer. If the comparator output is 0, then the multiplexer output is the value $C(k-1, l-1)$, and otherwise it is $C(k-1, l-1) + \delta$. This implements the required computation in one clock cycle.

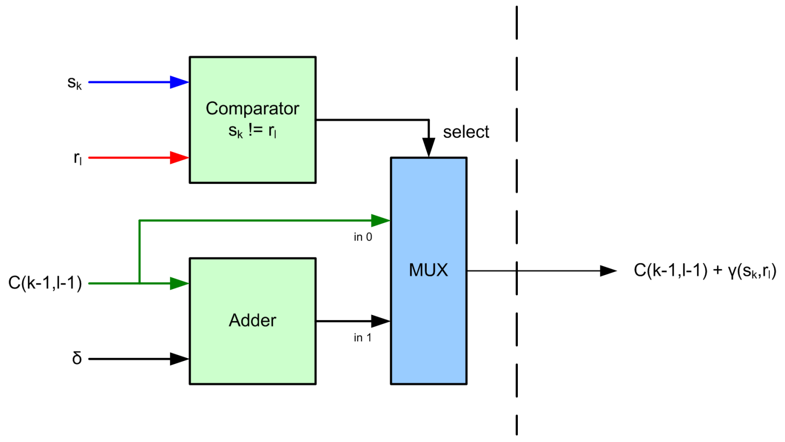

Figure 3.6: Subsystem to Compute $C(k-1, l-1) + \gamma(s_k, r_l)$ in one clock cycle.

Figure 3.7 shows the design of the subsystem to compute the minimum of the three values $C(k, l-l) + \alpha$, $C(k-1, l) + \beta$, and $C(k-1, l-1) + \gamma(s_k, r_l)$. The implementation uses two instances of a design to compute the minimum of two values. Each of these instances takes one clock cycle to complete, so the whole computation takes two clock cycles. The first instance calculates the minimum of $C(k, l-l) + \alpha$ and $C(k-1, l) + \beta$. This intermediate result is then compared in the second half of the design against $C(k-1, l-1) + \gamma(s_k, r_l)$, and hence the output is the minimum we desire.

Figure 3.8 shows the low-level detail of a full Processing Engine. The subsystems that we have introduced in figures 3.6 and 3.7 are marked as Subsystem A and Subsystems D and E respectively. Subsystems B and C compute the "left" and "up" dependent scores, $C(k, l-l) + \alpha$ and $C(k-1, l) + \beta$ respectively.

Once the system has computed the value for cell $(k, l)$, it is stored in the register `last` at the start of the next computation. Similarly, at the start of the next computation the value in register `lastlast` is set to the current value in register `last`.

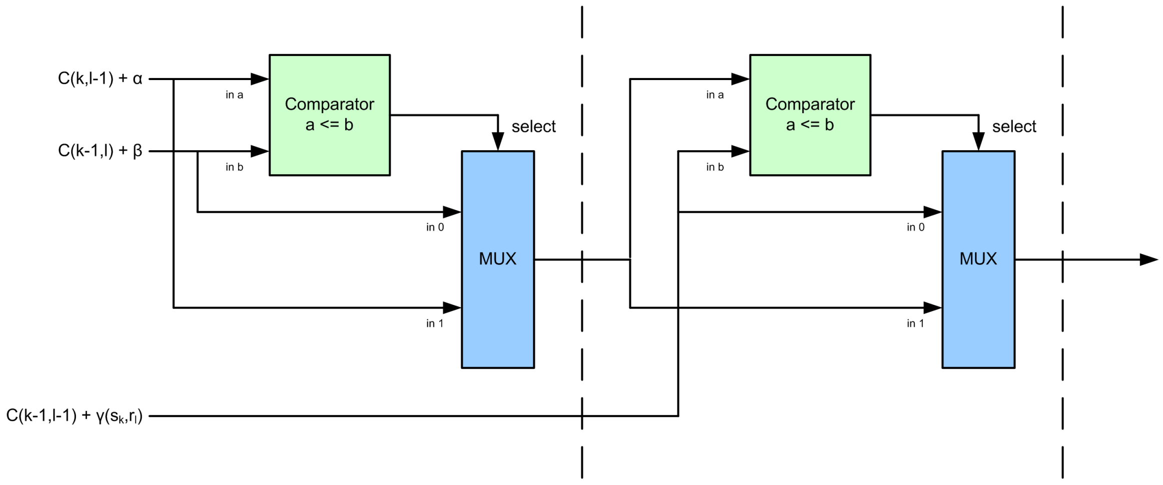

Figure 3.7: Subsystem to Compute Minimum of Three Values in two clock cycles. The inputs $C(k, l-l) + \alpha$, $C(k-1, l) + \beta$, and $C(k-1, l-1) + \gamma(s_k, r_l)$ are supplied, and thus the output $C(k, l)$ is computed.

This behaviour is regulated by an "Enable" input. The enable input is a signal that is set high (i.e. "true") for one clock cycle every four clock cycles.

The "Reset" should only be set high for one clock cycle for the duration of a whole alignment. A reset forces the register `last` to be set to value 0. When the processing engines are connected together, a single reset pulse is sent at the start of the computation, and it is propagated through all the processing engines, with a delay of 4 clock cycles between each PE. This is necessary to properly initialize the registers in each PE at the appropriate time, while the pipeline is being filled.

### 3.1.3 A Stand-alone Single Alignment Implementation

The FPGA design for a single alignment implementation in a single FPGA is shown in Figure 3.9. We need to support reference lengths of more than 100 million base-pairs. Since the representation of each base-pair requires at least 2 bits, a 100 million long reference sequence requires at least 23.84MB of storage. Clearly this is not achievable in block RAM on the FPGA, so the BEE2's onboard DRAM needs to be used. Hence the figure shows that the systolic array receives the stream of reference sequence base-pairs from DRAM. A DRAM interface block is necessary to convert the DRAM output into a stream of 2-bit values. It is important that this stream be

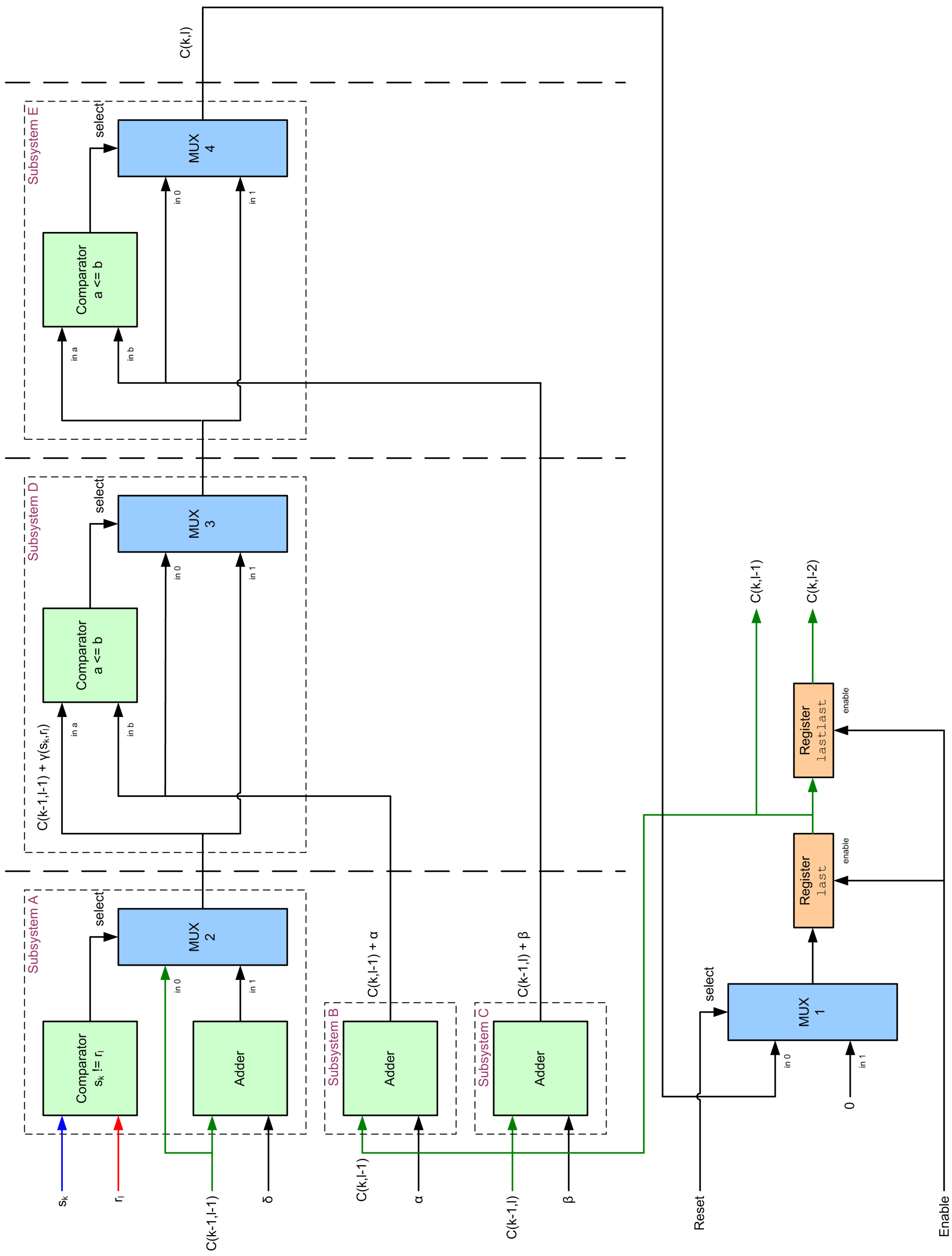

Figure 3.8: Low-level Details of a Processing Engine. The subsystems for computing scores and minima are marked A through E. The feedback of the results to registers, including reset and clocking (enable) logic is also shown.

continuous (since the systolic array does not allow for gaps in its inputs) and easily controllable (so that the sequence can be quickly rerun for the execution of a new short-read alignment). We devoted considerable effort to building and testing this subsystem, which we detail in the next subsection.

The systolic array's output is provided to a subsystem that determines if the score should be reported. We only wish to report the location in the reference of alignments where the score for the alignment is below a user-specified threshold. If a score is selected by the thresholding subsystem, the current location in the reference sequence is written to a FIFO buffer that is accessible from the PowerPC. A program (written in C) running on BORPH is used to read out the buffer once the alignment computation is complete. (We only expect between 5 and 10 scores to be below the threshold for an alignment against a 100 million base-pair reference sequence, so a FIFO buffer depth of 512 is sufficient for all practical purposes.)

Figure 3.10 shows the threshold logic at the end of the systolic array, and the shared FIFO buffer that it feeds into. The thresholding comparator returns true if the computed score is below, or equal to, a user-specified threshold. Conceptually all that is necessary is for this output to be connected to the "write enable" input of the FIFO buffer. However, there are a couple of subtleties that need to be taken into account.

Firstly, for the first $4 \cdot M$ clock cycles[9] of the operation of the systolic array, the output score will be invalid, since the pipeline is still being filled. After $4 \cdot M$ clock cycles have elapsed, the pipeline is full, and each emerging score from that point on is valid. In Figure 3.10, the second comparator is used to ensure that the first $M$ scores are ignored. (The method we use to avoid missing a genuine match in the first $M$ base-pairs is to pad the reference sequence with $M$ dummy base-pairs at the beginning of the sequence.)

Secondly, since we only receive a new score from the systolic array every 4 clock cycles, we should only write to the FIFO when a new score becomes available. The counter with the positive edge detection in figure 3.10 is used to implement this. The

[9]Each cell update requires 4 clock cycles to be computed, and we need to wait for all $M$ processing engines to have been initialized.

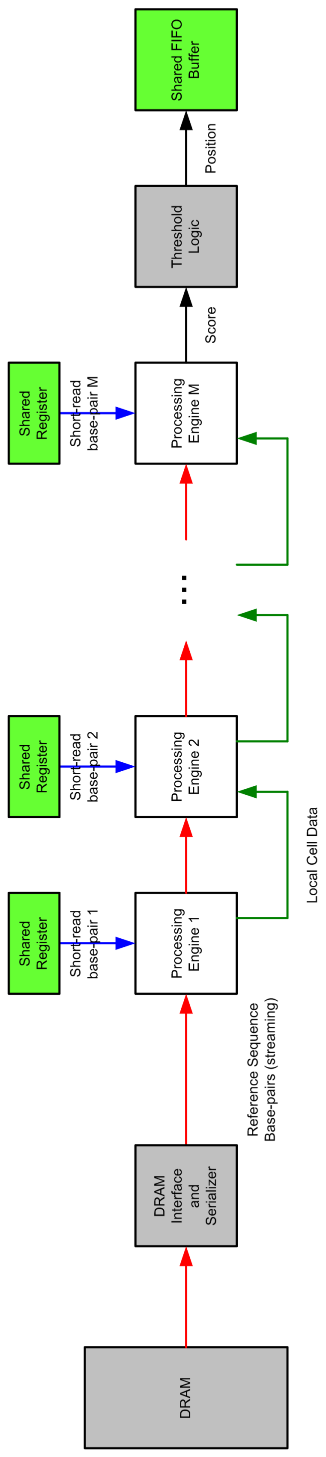

Figure 3.9: A Stand-alone Single Alignment Design. The systolic array is shown being supplied with a stream of reference sequence base-pairs from DRAM. The short-read base-pairs are supplied by shared registers, which are accessible in the BEE2's PowerPC. The reference sequence position of scores that are below a user-specified threshold are stored in a FIFO that is also visible to the PowerPC.

counter is incremented only every 4 clock cycles. On the clock cycle that it increases, the positive edge detection will output true.

If these conditions are satisfied, then the "write enable" input to the shared FIFO buffer is brought high for one clock cycle. The data input is connected to a counter, whose value can be used to deduce the position in the reference sequence. This counter is reset when the computation of a new alignment is begun. The same reset signal is used to reset the shared FIFO buffer. The systolic array is always computing, so the definition of the start of an alignment is when the reset signal is asserted. This same reset signal is also used to restart the readout of the reference sequence from the DRAM.

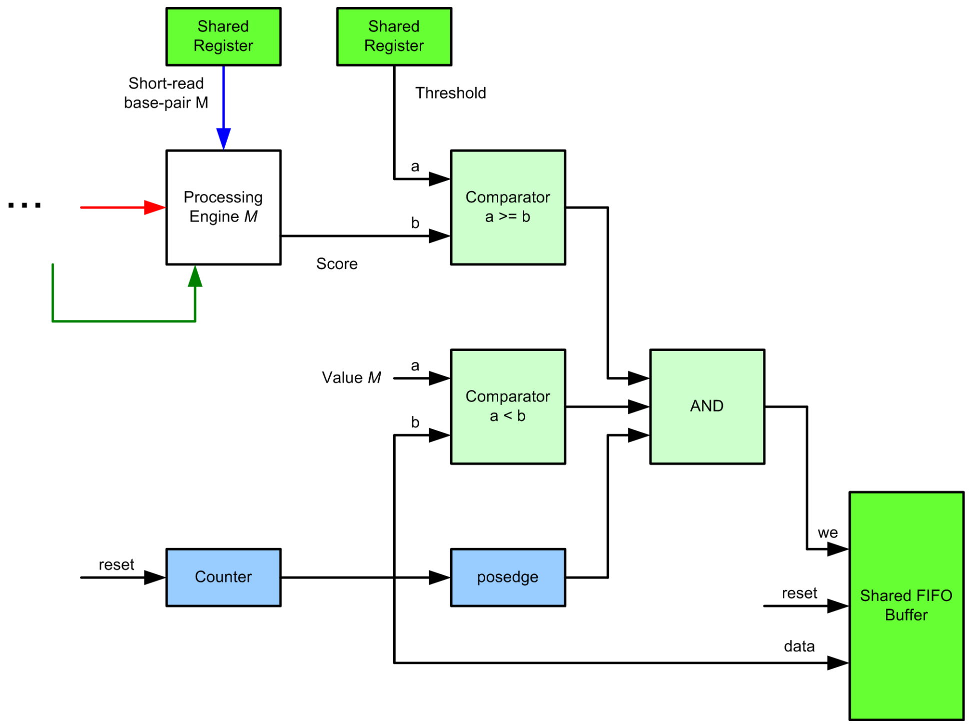

Figure 3.10: Score Threshold and FIFO Buffer Output Stage. The score output from the final processing engine in the systolic array is passed to a thresholding output stage that conditionally writes the corresponding reference base-pair index to a FIFO buffer if the score is below a user-specified value.

### 3.1.4 DRAM Interface for Reference Streaming

To support long reference sequences, we need to allow the reference sequence to be stored in DRAM. A DRAM controller typically has a wide data bus, but the systolic array requires a stream of 2-bit reference base-pair values. Thus DRAM support necessitates the development of not only an interface to DRAM, but also of a serializer (or "parallel-to-serial converter"). This was shown in Figure 3.9.

Specifically, we desire to provide the systolic array with a 2-bit value every four clock cycles. Therefore we need only obtain one 64-bit word from the DRAM every 128 clock cycles. It is, however, important that the stream of reference base-pairs to the systolic array never be interrupted – for this reason, we carefully designed the DRAM interface and serializer to use a double-buffering scheme that ensures a steady stream of data.

Figure 3.11 shows the DRAM interface and serializer design that we implemented. We allow for the storage of up to $2^{27}$ 64-bit words, where each word contains 32 2-bit reference characters. Hence, reference sequence lengths of $2^{27} \times 32 = 2^{32}$ base-pairs are possible. The BEE2 Simulink toolflow-based DRAM controller was used as the DRAM abstraction. The core functionality of our interface is the logic to request words from the DRAM at appropriate intervals.

Once a word is read from DRAM, it is immediately written to the first port of a dual-ported BRAM[10]. A clock divider is used to produce a pulse every four clock cycles, which in turn clocks a counter that cycles through the 64 possible addresses on the second BRAM port. The next 64-bit word from the DRAM is requested and written to the dual-ported BRAM as soon as possible. Since the read-latency of the DRAM is guaranteed to be less than 128 clock cycles, the double-buffer will always be able to supply the systolic array with an uninterrupted data stream.

[10]The BRAM has a total size of 128 bits. The first port uses a 64-bit data line, with two valid addresses, and the second port uses a 2-bit data line, with 64 valid addresses.

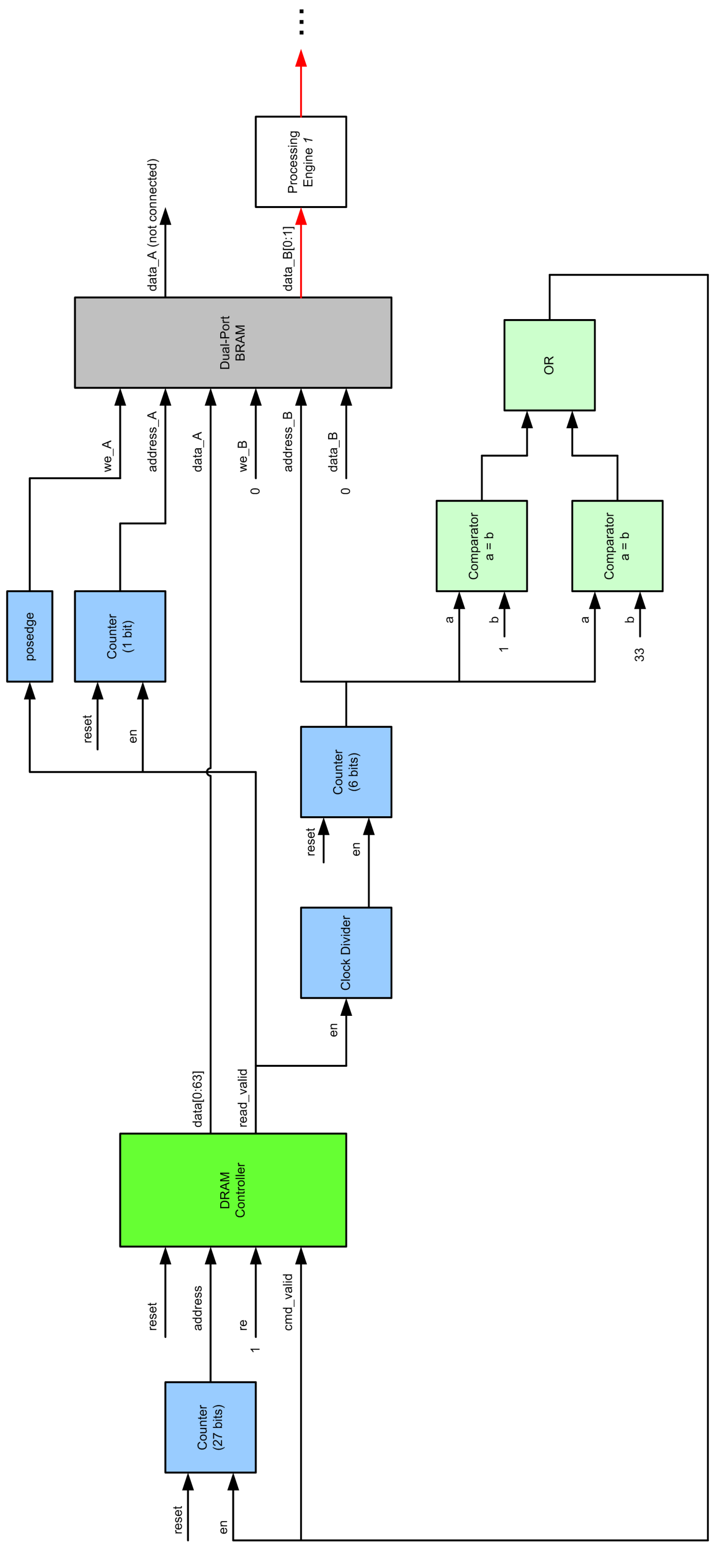

Figure 3.11: DRAM Interface and Serializer. The DRAM controller reads 64-bit words which are written to a double-buffer implemented in a dual-ported BRAM. The BRAM is used as the serializer by defining its second port to have a data width of 2 bits. This design allows the systolic array to be provided with an uninterrupted stream of reference characters.

## 3.2 A Parallel Multiple Short-read Alignment Implementation

A single systolic array computes a single alignment of a short-read against a reference sequence. However, it is possible to fit several independent systolic arrays onto an FPGA, and by doing so it is possible to align several short-reads in parallel. The extension of the design from a single alignment processor to one that can perform $P$ alignments in parallel is shown in Figure 3.12. The systolic array is simply replicated as many times as we can fit it onto a single FPGA. The DRAM interface and serializer feeds all the systolic arrays in parallel.

The key difference between the multiple alignment design and the single alignment design is the output stage. Whereas the single alignment design (c.f. Figure 3.9) uses a shared FIFO buffer to store the alignment results, in this multiple alignment design we do not simply replicate $P$ shared FIFOs and associate each FIFO with a systolic array. We avoid this technique primarily because shared FIFOs use up significantly more FPGA resources than do ordinary FIFO buffers[11]. Instead, each systolic array writes its results to a regular FIFO buffer. A "round-robin" reader cycles through these FIFOs and writes their contents to a single shared FIFO buffer. This shared buffer is the one with which the user interacts.

### 3.2.1 Round-robin FIFO Output Stage

Figure 3.13 shows the details of the thresholding and non-shared FIFO buffer stage that is connected to the output of each systolic array. The output stage is very similar to that for a single alignment design (c.f. Figure 3.10). The primary differences are that the shared FIFO buffer has been replaced with a regular FIFO buffer, and there now exists some logic to perform reads from the FIFO. (In the single alignment version, the FIFO is only read through the PowerPC, and not within the design.) Specifically, there are two inputs for supporting reads, `index` and `select`. If `index` = `select`, then a value from the FIFO will be read (if the FIFO is not empty). The `index` input value is concatenated with the FIFO data output and empty signal.

[11]Shared FIFO buffers need to be attached to a bus through which the PowerPC, and hence BORPH, can communicate with them. It is this bus connection that is resource-intensive.

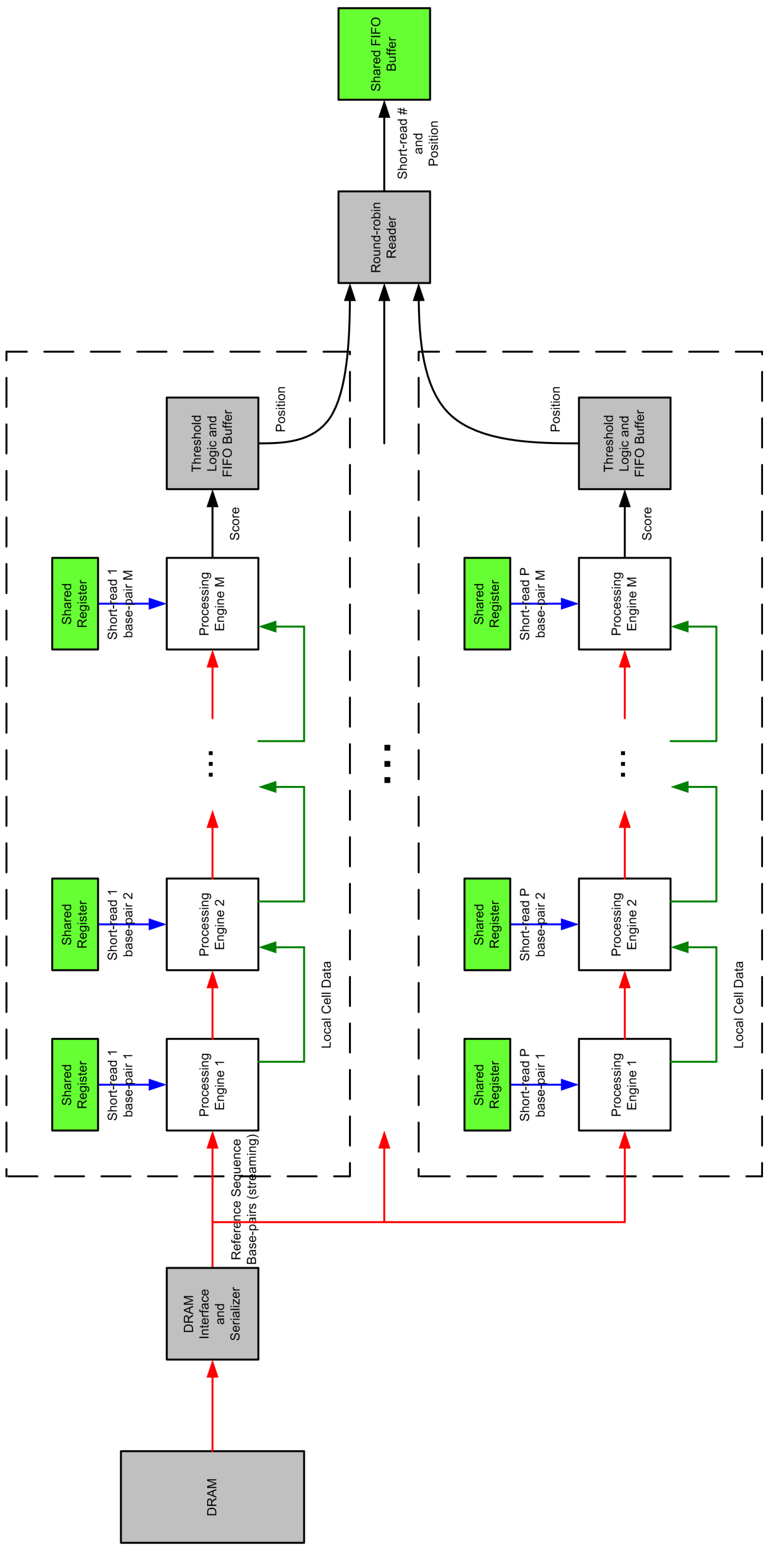

Figure 3.12: A Parallel Multiple Alignment Design. $P$ copies of the systolic array design presented in Figure 3.9 can be placed in a single FPGA, to allow the parallel computation of $P$ different short-read alignments.

This concatenated value is the output of the stage.

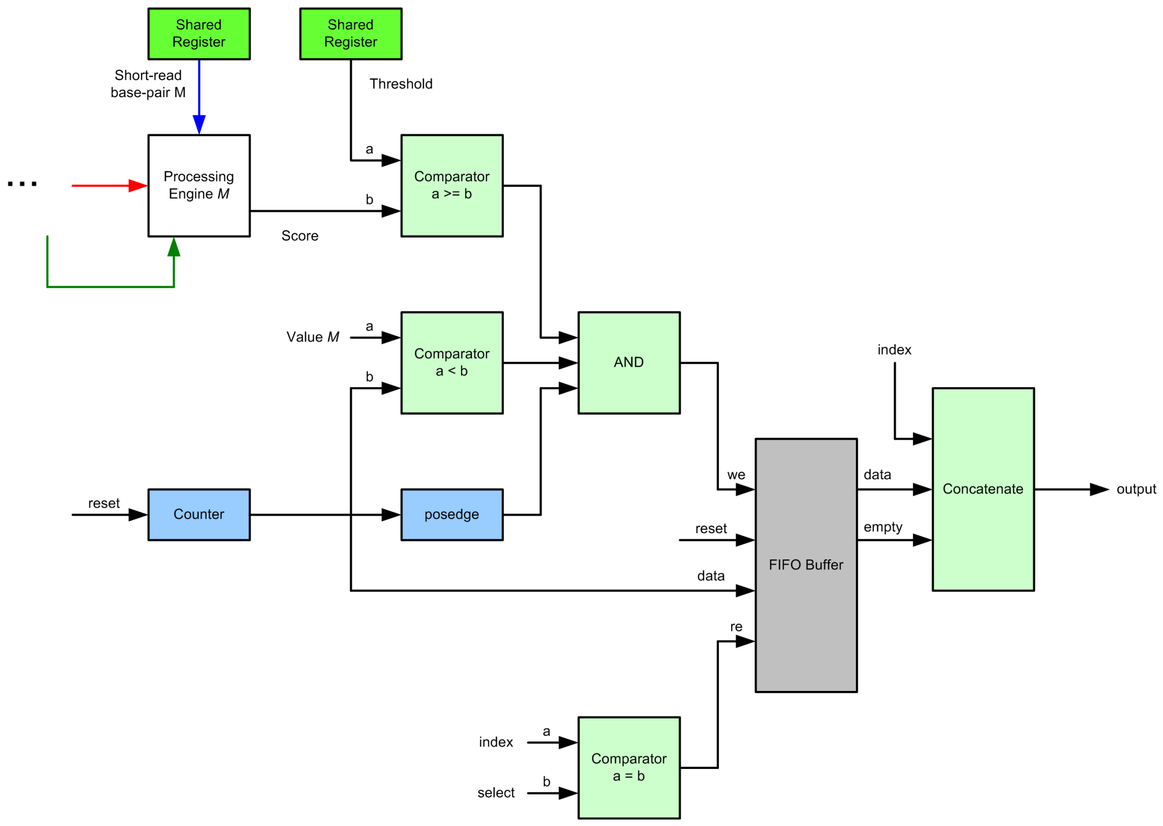

Figure 3.13: Score Threshold and FIFO Buffer Output Stage.

Figure 3.14 shows the round-robin reader. This subsystem is designed to continuously cycle through the FIFOs at the end of each systolic array, read their values if they are not empty, and write those values to a single shared FIFO. A counter is used to generate a repeating sequence $0, 1, \ldots, P-1, 0, 1, \ldots$. This counter value is used to "select" a particular FIFO, and perform a read from it. To allow this, each alignment's output stage is assigned an index value, where the first systolic array has index 0, the second index 1, through to the $P$th, which is assigned index $P-1$. The counter value is used to select a particular input on multiplexer, so the multiplexer's output cycles through the outputs of systolic array buffer stages.

When a FIFO is read, it outputs an `empty` signal that is true if the FIFO buffer was empty. This signal is included in the output of each systolic array's buffer stage. The round-robin reader therefore needs to unpack the data to check this `empty` value. Data is only written to the shared FIFO if `empty` is false. Both the alignment index and the data are written to the shared FIFO – this is necessary for the user to determine which alignment the result is from.

The use of a round-robin readout limits the rate at which data can safely be produced. If each systolic array produces output at a rate greater than $1/P$ matches per clock cycle, the FIFO buffers will overflow, and results will be lost. However, such high output data rates are not expected – the thresholds are chosen such that each short-read alignment produces of order 10 match results in a 3 billion base-pair reference. Furthermore, bursty output can be handled without loss since the FIFOs are deeper than the size of the largest expected result set.

## 3.3 Parallelization Across Multiple FPGAs

Since multiple sequence alignment for resequencing is an embarrasingly parallel problem, the extension of a single-FPGA implementation to a many-FPGA system is conceptually simple. If we have $Q$ short-reads that we need to align, and $N$ FPGAs at our disposal, we simply assign $Q/N$ short-reads to each FPGA. The computation time for alignment is constant in the case where the short-read length $M$ and the reference sequence length are constant, hence this simple work allocation scheme is adequate.

The practical details of extending the system beyond a single FPGA primarily concern the development of control software to manage the computation on a cluster of BEE2 boards. Each BEE2 board contains four user FPGAs, and one control FPGA that runs BORPH. Each of the four user FPGAs is run entirely independently of the others. There is only a small difference in the cluster control software required to manage FPGAs on different boards versus FPGAs on the same BEE2.

Figure 3.15 shows the architecture of our BEE2 cluster. Each BEE2 operates independently, but is connected over 100MbE to a switch, to which a control terminal is attached. Inside each BEE2 the control FPGA's DRAM contains a copy of the reference sequence. This sequence is distributed to the DRAM directly attached to each user FPGA. The control FPGA communicates with each user FPGA over high-speed serial links – these communications include the setting up of short-read sequences,

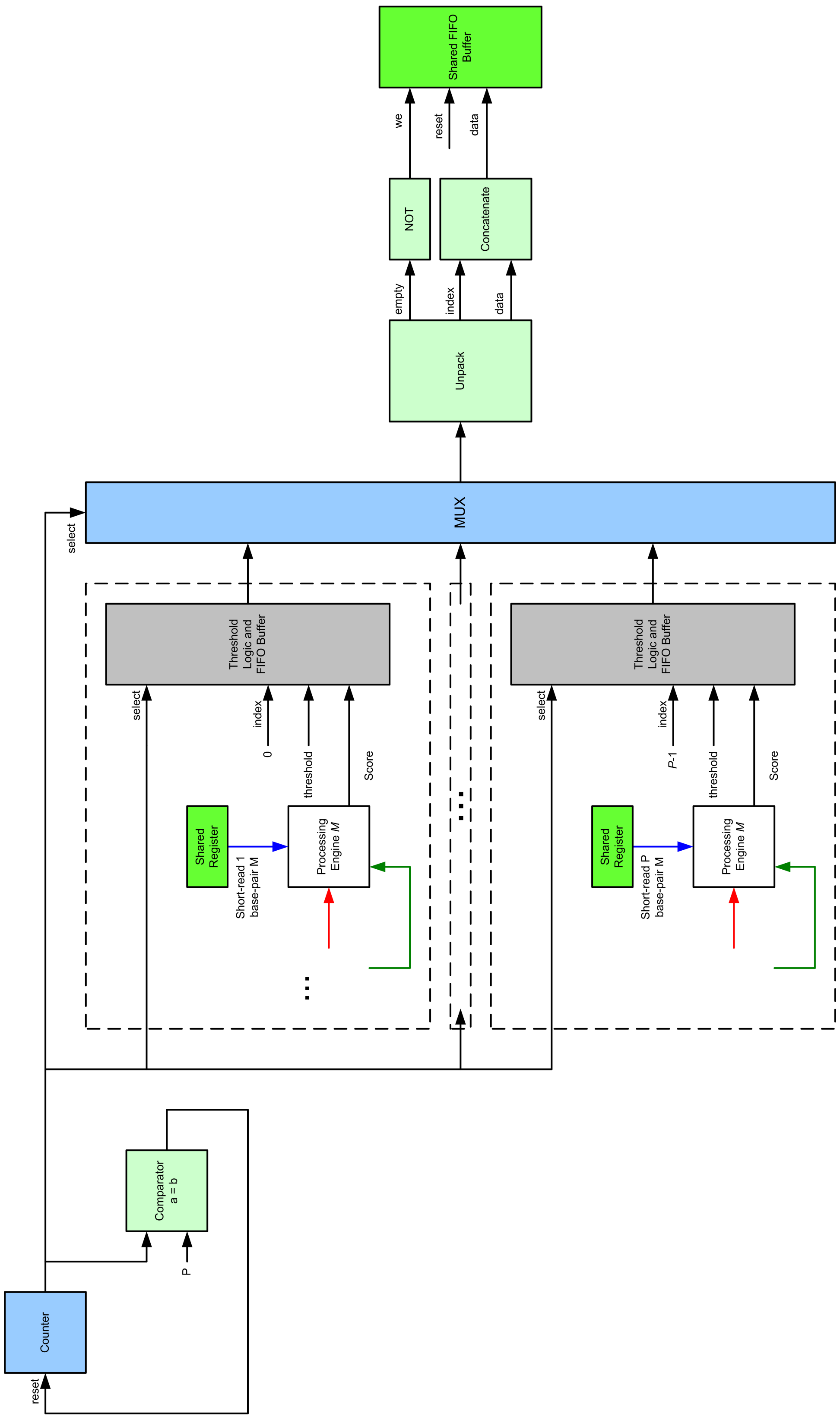

Figure 3.14: Round-robin FIFO Reader and Shared FIFO Output Stage.

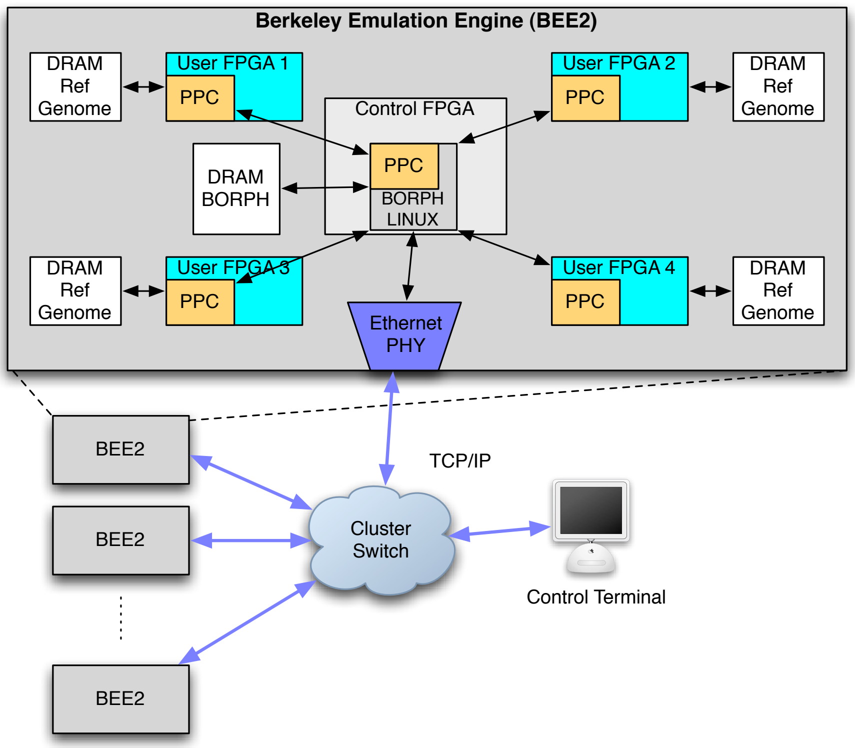

Figure 3.15: BEE2 Cluster Architecture. Figure courtesy Vinayak Nagpal.

and the reading out of alignment results. Both the short-read inputs and the outputs for all four FPGAs are channeled through the control FPGA, which in turn communicates with the control terminal over the 100MbE control network.

Figure 3.16 shows the flow of the control software for the cluster. The control terminal (which acts as the "head node" in cluster computing parlance) is primarily responsible for dividing the $Q$ short-reads into $N$ distinct sets, where each set is intended for processing on a single FPGA. The head node distributes the reference sequence to each BEE2 control processor, which in turn loads the reference into the DRAM of each user FPGA on that BEE2.

Each FPGA can process $P$ short-read alignments in parallel. Each FPGA is allocated $Q/N$ short-reads. Therefore each FPGA must process $\frac{Q/N}{P} = \frac{Q}{N \times P}$ separate batches of short-reads (with each batch containing $P$ short-reads). The control processor on the BEE2 divides each FPGA's allocation into $\frac{Q}{N \times P}$ batches. For each batch, the control processor loads the queries into the FPGA's shared registers, starts the computation, waits for the computation to complete, and reads the results from the shared FIFO. The results are then saved to disk on the head node.

Having presented the design and implementation of a parallel sequence alignment cluster, in the next chapter we provide results that demonstrate its performance advantage over conventional software implementations.

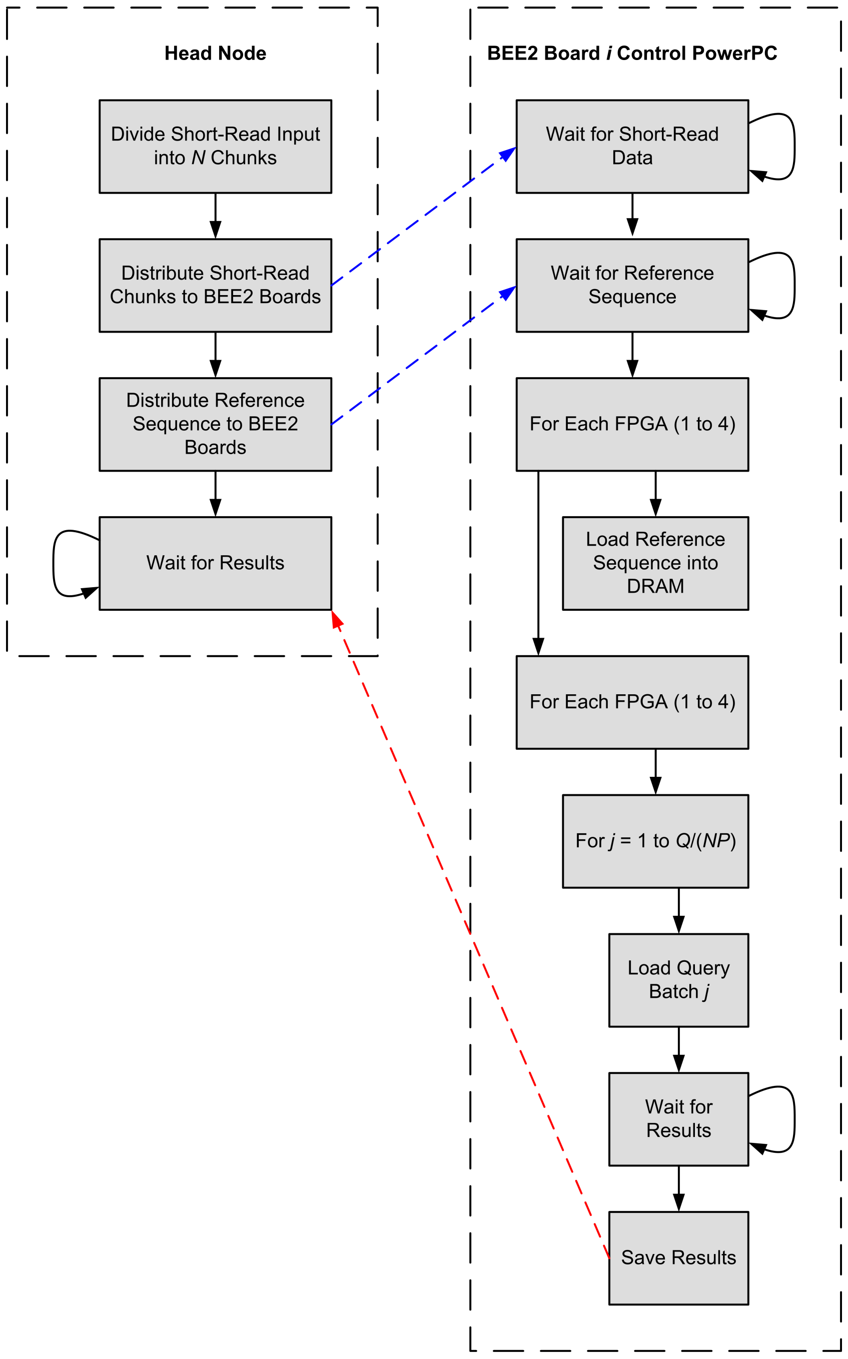
Head Node
Divide Short-Read Input into N Chunks
Distribute Short-Read Chunks to BEE2 Boards
Distribute Reference Sequence to BEE2 Boards
Wait for Results
BEE2 Board i Control PowerPC
Wait for Short-Read Data
Wait for Reference Sequence
For Each FPGA (1 to 4)
Load Reference Sequence into DRAM
For Each FPGA (1 to 4)
For j = 1 to Q/(NP)
Load Query Batch j
Wait for Results
Save Results

Figure 3.16: Cluster Control Software Flow.

# Chapter 4

# Results and Discussion

In this chapter we report our results from testing both our single FPGA and 32-FPGA sequence alignment implementations.

## 4.1 Single FPGA

In this section we present two sets of results. The first relates to the number of sequences it is possible to align in parallel (i.e. the value $P$) for particular short-read sequence lengths. Clearly as the short-read length increases, it will only be possible to simultaneously align fewer sequences, since each cell update processing engine uses a set amount of FPGA resources. The second set of results we present are the actual performance results obtained during the benchmarking of our system. It is possible to calculate a theoretical maximum performance for the system based on the FPGA clock speed and the number of processing engines placed on a single FPGA, but this neglects the start-up cost of setting the reference sequence and short-read inputs, and of reading the output when the computation is finished. We benchmarked the performance of the FPGA, including all overhead, and compared our results to the performance on a state-of-the-art optimized CPU software implementation.

### 4.1.1 Correctness

Our first test of the FPGA system was designed to determine the correctness of the system. Specifically we loaded a 100 million base-pair *Drosophila melanogaster* reference sequence into DRAM, and performed alignments against 100 different short-read sequences each of length 31 from [5]. We compared the results to that of a conventional CPU implementation (written in C) and found that our FPGA results exactly matched those from the CPU.

### 4.1.2 Short-read Length Resource Scaling

As we have mentioned in Chapter 2, current short-read sequencing technology produces short-reads of length approximately 30 base-pairs and this value is likely to double in the next few years [9]. In this chapter we benchmark the performance of our system for $M = 31$, and we also demonstrate that our design will scale well with longer short-reads. The latter result is now described in detail.

Table 4.1 shows how the number of systolic arrays that can be packed into a single FPGA design scales with the length of the short-reads $M$. Since each systolic array contains $M$ processing engines, the total number of PEs is $M \times P$. The greater the number of PEs packed, the better the performance is, since each PE computes a cell update every four clock cycles. Hence a larger number of PEs packed results in a larger number of cell updates that are done in parallel.

One reasonable line of questioning about the scalability of our design and architecture concerns the limits – the cases where $M \ll 31$ and $M \gg 75$. These are not of great practical interest, since short-reads produced by current sequence technology have length approximately 30, and future advances are not expected to soon extend these lengths much beyond 75. Nevertheless, it is instructive to consider these questions. Our results show that it is possible to use our design for computations where the short-read length $M = 75$, and suffer no degradation in performance versus the cases where $31 < M < 75$. In fact, the number of processing engines packed when $M = 75$ is greater than for $M = 31$.

For the case $M \gg 75$, the overhead per systolic array will be decreased. By "overhead", we mean the necessary parts of the design that are not directly performing dynamic programming table cell updates (processing engines). The most significant overhead per systolic array is the threshold and FIFO buffer output stage. With overhead excluded, we expect that a particular FPGA will be able to fit a constant number of processing engines. Therefore, if we increase the short-read length $M$, we will have to reduce the number of systolic arrays packed, $P$. This results in less overhead per processing engine, and thus in a more efficient design.

The most efficient use of the FPGA is therefore the case where $P = 1$ and $M$ is set to the largest value that yields a design that will fit in the FPGA. Since $M$ is not a flexible parameter, this fact is of no immediate practical use to us. However, Table 4.1 indicates that the maximum number of processing engines that can be fitted on a Xilinx Virtex 2 Pro VP70 FPGA is at least 630 (when $P = 1$). There is overhead associated with each systolic array, so the highest achieve PE count when $P > 1$ is a lower bound. Therefore our design is conservatively limited[1] to cases where $M < 630$.

There is a potential disadvantage to designs for the large-$M$ cases: since $P$ has to be a whole number, there are many values of $M$ where the product $M \times P$ is up to one half as small as it could be were fractional values of $P$ allowed. For the sake of argument, assume that when $M = 650$, we can fit one systolic array onto the FPGA (i.e. $P = 1$), and that this is optimal ($M$ cannot be increased). If we wish to make a design to perform alignment of short-reads where $M = 330$, we will still only be able to fit one systolic array, and not two. Therefore a large amount of FPGA resources would be left unused. (Of course it is possible to alleviate this problem by making systolic arrays with different lengths, but this is not desirable to the user, who has constant-length short-reads.)

Fortunately, this issue of granularity is not particularly severe when $31 \leq M \leq 75$, since $8 \leq P \leq 18$. In fact, the case $M = 31$ suffers from this inefficiency ($M \times P = 558$ in this case, whereas typically $M \times P \approx 600$). However, when $P > 10$, rounding down to the nearest integer in the worst case results in an efficiency loss of less than 10%. Table 4.1 clearly shows that for the cases relevant to our application ($31 \leq M \leq 75$), this potential loss of efficiency in larger-$M$ cases is not significant.

For the case $M \ll 31$, the analysis is simpler: the overhead from the DRAM interface and FIFO output stage becomes increasingly large in comparison to the re-

[1]This empirical limit applies to the Virtex 2 Pro VP70 FPGA. Of course the limit will be increased for FPGAs that contain more slices, such as the Virtex 5 LX330T FPGA that has approximately 4X more resources.

Table 4.1: Dependence of resource utilization on the length $M$ of short-reads. For a given $M$, we tried to pack as many short-reads of length $M$ as possible on a single FPGA. Table courtesy Henry Chen.

| Short-read length $M$ | Number $P$ of systolic arrays packed | Total PEs ($M \times P$) |
|---|---|---|
| 15 | 34 | 510 |
| 31 | 18 | 558 |
| 45 | 14 | 630 |
| 60 | 10 | 600 |
| 70 | 9 | 630 |
| 75 | 8 | 600 |

sources used by the processing engines, and hence the efficiency of the FPGA design dramatically decreases. Fortunately this is not a concern for our application, where $M$ has in practice a lower bound of approximately 30.

Figure 4.1 most clearly shows our claim that our design scales as the short-read length increases from 31 to 75. In ideal scaling (where we do not consider the aforementioned effects of granularity and overhead), we expect the total number of PEs to remain constant. The graph shows this to be approximately the case in practice, hence the performance of our system will not degrade as the short-read length is increased.

### 4.1.3 Performance

Performance of sequence alignment implementations is often conveniently measured in "cell updates per second" (CUPS) – a single cell update is a single application of the dynamic programming table cell update rule. This measure is useful for comparing implementations, since it is not dependent on the lengths of the two sequences being aligned. Modern computing systems can perform millions or billions of cell updates per second, so the units MCUPS and GCUPS are also frequently used. These respectively represent $10^6$ CUPS and $10^9$ CUPS.

To measure the performance of our implementation, we used a 100 million base-

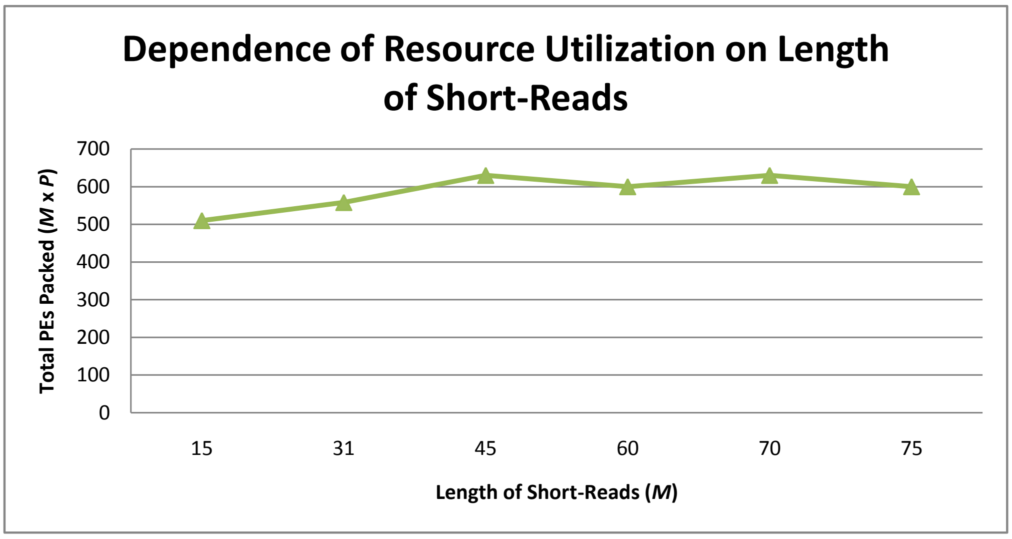

Figure 4.1: Graph Showing Resource Utilization as a function of Short-Read Length $M$.

pair *Drosophila melanogaster* sequence as the reference sequence and aligned varying numbers of length $M = 31$ short-reads against this reference.

Table 4.2 reports the performance of our implementation on a single FPGA. We report the "total time" in addition to the "compute time". The latter excludes once-off startup costs. Loading the reference genome took about 53.4 seconds, and other startup operations took 6.6 seconds. The reported GCUPS figures are based on the compute time (which includes the time to load the queries, perform the alignment and read out and store the results).

The loading of the reference genome is particularly slow due to a bottleneck in the interface between the PowerPC (on which BORPH runs) and the DRAM controller implemented in the FPGA fabric. A 100 million base-pair sequence has size 23.84MB (assuming 2-bit encoding). The maximum supported DRAM usage is 1GB, which allows reference lengths of greater than 4.2 billion base-pairs to be used. In practice, the alignment computations are expected to require several days of computation time, so even the time taken to load a 4 billion base-pair reference sequence would be insignificant compared to the total running time.

The time spent by the FPGA computing each alignment is deterministic, and is dependent on the FPGA clock speed $f_c$, the short-read length $M$ and the number of systolic arrays $P$. Specifically, the theoretical peak performance in CUPS is $\frac{f_c}{4} \times M \times P$, where $f_c$ is in units of Hz[2]. For a 200MHz clock speed, with $M = 31$ and $P = 18$, the theoretical peak performance is 27.9 GCUPS. This is unachievable in practice, since it neglects the time required to load the short-reads into the FPGA, and the time taken reading out the results. Table 4.2 shows that we were nevertheless able to achieve performance very close to this theoretical peak, and that hence the overhead in our system is not significant in practice.

[2]The division of the clock frequency by four is required because each cell update requires four clock cycles to complete.

Table 4.2: Performance of our implementation on a single FPGA on the BEE2 system. Loading the reference genome took about 53.4 seconds, and other startup operations took 6.6s. The total time shown below includes these startup times. The reported GCUPS performance excludes startup time, which is insignificant compared to runtime for jobs with many queries. The reported times are averages from 5 runs.

| #Short-Reads Aligned | Total Time (s) | Compute Time (s) | GCUPS |
|---:|---:|---:|---:|
| 900 | 163.5 | 100.5 | 27.0 |
| 1800 | 271.3 | 201.0 | 26.4 |
| 3600 | 471.2 | 402.0 | 27.1 |
| 7200 | 885.1 | 804.0 | 27.1 |
| 14400 | 1701.6 | 1608.0 | 27.2 |
| 28800 | 3342.4 | 3216.0 | 27.2 |
| 57600 | 6623.5 | 6432.0 | 27.2 |

**Performance Comparison with a State-of-the-Art Software Implementation**

There has been considerable effort in the software community to develop fast implementations of dynamic programming-based sequence alignment alogorithms. The current leading technique by Farrar [54] uses SSE2 instructions in Intel microprocessors to parallelize the computation of Smith-Waterman alignment. Terry Filiba [42] applied Farrar's technique to the optimized alignment algorithm used in this thesis, thereby producing a state-of-the-art software implementation that is directly comparable to our FPGA implementation.

Figure 4.2 shows the performance of Farrar's code when it is used to compute alignments of many length 31 short-read sequences against a 100 million base-pair reference sequence. A dual-processor quad-core 2GHz Intel Xeon workstation was used as the benchmarking platform[3]. Only a single CPU core was used in our benchmark[4]. The performance of Farrar's code degrades significantly as the short-read

[3]Our test machine used two Intel Xeon E5405 processors. These processors each have four cores, and a shared 12MB L2 cache. Their nominal power consumption is 80W (each). All software implementations were compiled using `gcc` with the `-O2` optimization switch.

[4]Due to the embarrasingly parallel nature of the problem, it is reasonable to assume a linear

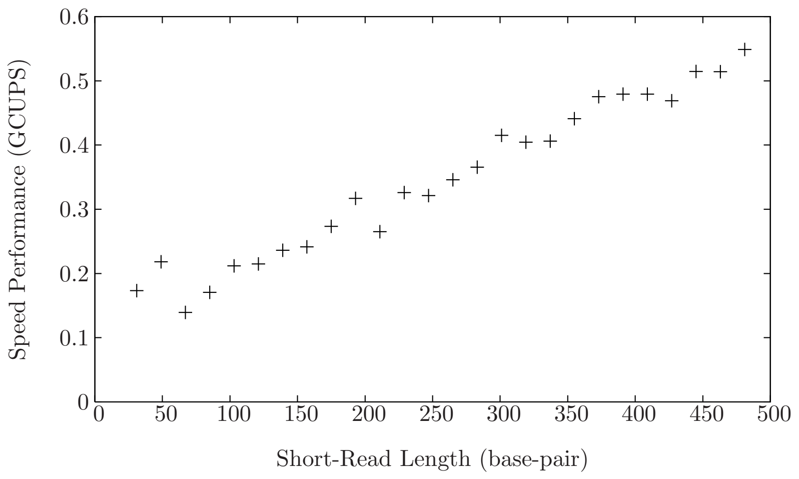

Figure 4.2: Speed performance (in GCUPS) of Farrar's code as a function of short-read length. Figure courtesy Terry Filiba and Yun Song.

length decreases: for short-reads with length in the range $25 < M < 100$, the performance is approximately 200 MCUPS (per CPU core).

A non-optimized CPU implementation of our algorithm, benchmarked using the same data, was able to compute 98 MCUPS. Filiba's SSE2-based code improved this to 447 MCUPS. The short-read length was set to $M = 31$ for all our performance benchmarks (on FPGAs and on CPUs), since this is the value for which the SSE2 parallelization has maximum efficiency. Therefore we provided the software implementation with the best possible conditions.

The single FPGA speed-up over a single CPU core is 61X. Therefore 61 CPU cores give the same performance as a single FPGA. An alternative implication is that a resequencing task requiring 6,100 hours for alignment would be reduced to only 100 hours if a single FPGA were used instead of a CPU. This represents a dramatic speed-up that is surely of interest to resequencing researchers.

We estimated the power consumption of the Xeon workstation to be 200W[5].

speedup with the number of cores. This enables a system-level comparison with our FPGA implementation, and in particular allows us to estimate the number of CPUs that would be required to achieve equivalent performance to a single FPGA.

[5]This figure considers the power consumption of the entire machine, including both CPUs (nominally 80W each).

The power-performance of this workstation running one query per core is 0.01788 GCUPS/Watt[6]. We provide a power performance comparison with the FPGA system in the following section.

## 4.2 Cluster

To test the performance of the cluster implementation, we ran an alignment of 460800 short-reads (each with length 31 base-pairs) against the same 100 million base-pair *Drosophila melanogaster* reference sequence as was used in the single FPGA benchmarks.

The fixed startup cost for the cluster implementation is larger than that for the single FPGA implementation – it is approximately 240 seconds. This increase is primarily due to the fact that each BEE2 needs to sequentially load the reference sequence four times. There is also a small contribution to the overhead from network communications tasks that do not occur in the single FPGA implementation.

We performed the tests on a cluster[7] of eight BEE2 boards, each containing four user FPGAs. We successfully aligned 460800 short-reads in an average of 1667 seconds (excluding startup time). We ran the alignment test 5 times, aligning all 460800 short-reads in each case, and the runtimes (excluding startup time) all fell within 2 seconds of 1667 seconds. This implies a performance for the cluster of 857 GCUPS, which is a performance of 26.8 GCUPS per FPGA. Hence, we deduce that the system scales well to 32 FPGAs, since the performance of a single FPGA is 27.2 GCUPS. This is expected, since the algorithm is embarrasingly parallel.

During the execution of the BEE2 tests, we monitored the power consumption of the cluster, including the control computer used for scheduling computations, storing data (input and output), and loading sequences onto the reconfigurable computers. We found that the power consumption of the cluster varied between 750W and 850W. The power-performance of the reconfigurable computer cluster system is therefore at

[6]We calculated this value by assuming a linear speedup with the number of cores. Eight cores can compute $8 \times 0.447 = 3.576$ GCUPS, and the overall power consumed for the 8-core system is 200W, hence the power-performance measure of system is $3.576/200 = 0.01788$ GCUPS/W.

[7]We used the BEE2 cluster built for the RAMP Blue project [55].

least 1.01 GCUPS/Watt[8].

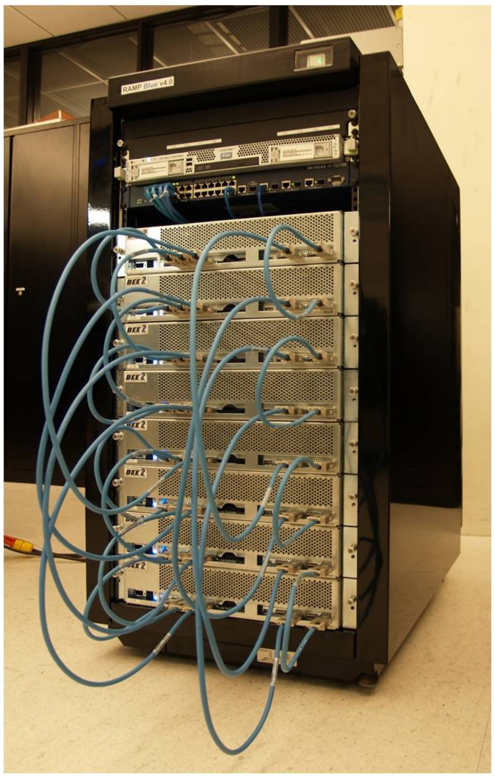

Figure 4.3: BEE2 Cluster. Photo courtesy Alex Krasnov.

The state-of-the-art software implementation running on a modern Intel Xeon-based system is capable of computing 0.01788 GCUPS/Watt. Based on the power performance results from the BEE2 cluster experiment, we determine that the FPGA implementation provides a 54X improvement in power-performance over its conventional software counterpart.

## 4.3 Conclusion

Using our custom alignment processor design, we achieved a 61X speedup for a single BEE2 FPGA versus a single modern CPU core. We tested our design on a 32-FPGA cluster and provided performance results showing linear scaling performance as the number of FPGAs is increased. Using our cluster implementation we were also able to demonstrate a 54X improvement in performance-per-Watt for our FPGA

[8]We used the upper bound of 850W in our calculation; 857/850 = 1.01 GCUPS/W.

implementation versus the best CPU implementation.

# Chapter 5

# Conclusions and Future Work

In this thesis, we have presented a custom FPGA design that accelerates a dynamic programming sequence alignment algorithm for genome resequencing applications by 61X versus a modern single CPU core. We have showed that the design scales linearly with a cluster reconfigurable computer system with 32 FPGAs. Further, we see no barriers to scaling the system to hundreds of FPGAs, since the alignment problem we solve is embarrasingly parallel and our system architecture has demonstrated no bottlenecks to prevent such scaling.

The promising results presented in this thesis suggest that there is much profitable work that remains in investigating the performance of short-read alignments on FPGAs, and in the development of production systems for use by biologists.

The most important near-term goal is the porting of our prototype design to Xilinx's Virtex 5 family of FPGAs. The BEE2 platform that we used is based on Xilinx's Virtex 2 Pro family, which is now over four years old. The Virtex 5 range makes improvements in four critical areas: they have more resources (FPGA slices), allowing for larger designs; they can be clocked at nearly double the frequency of Virtex 2 Pro chips; they use less power, and they are cheaper (per slice). The Virtex 2 Pro XC2VP70[1] FPGA used in the BEE2 has 4.4X fewer logic resources than the largest Virtex 5 chip, the XC5VLX330T. Therefore this Virtex 5 FPGA could support a design with more[2] than 4.4X the number of parallel short-read alignment systolic

[1]The full name of the part is: XC2VP70-7FF1704C.

[2]Since the design has some fixed overhead, primarily the DRAM interface, the increase in computing capability would be greater than the increase in capacity.

arrays. A Virtex 5 design would conservatively be able to run at a clock frequency of 350MHz, in contrast to the 200MHz possible with the Virtex 2 Pro. A 400MHz design may be possible[3]. Therefore a further improvement of at least 1.75X will be possible with a port to the Virtex 5. A total improvement in performance of 7.7X should therefore be relatively easily obtainable. This would push the advantage of the FPGA implementation over the CPU implementation to beyond 400X.

Each cell update processing engine in the systolic array design currently takes 4 clock cycles per computation. (Figure 3.8 shows the detailed design of a single cell update processing engine.) Therefore, the speed at which the systolic array operates is four times slower than the clock frequency of the FPGA. However, within the processing engine there are four separate stages, each of which only requires one cycle to execute. On any given clock cycle, one of the stages is performing useful work and three of the stages are lying idle — or more precisely, they are computing with garbage input, and their output is ignored. It is possible to modify this design to take advantage of the fact that on every clock cycle every stage is performing a computation. Specifically we[4] propose the pipelining of the internals of the cell update processing engines. This can be done relatively easily by assigning not one, but four short-read base-pairs to each cell update engine. A multiplexer should be placed at the front of the engine, and be used to cycle through the four short-read base-pairs. In this way it will be possible for each systolic array to execute four short-read alignments in the same time it takes for the current design to execute just one. This is a direct result of the effective clock rate of the systolic array increasing by a factor of four.

The FPGA design is very sensitive to changes in the bitwidths of variables. If it is possible to modify the algorithm to use fewer bits per cell score (or per penalty constant), dramatic improvements in performance can result, since the amount of resources that the systolic array requires is directly related to the bitwidth of the data pipeline it contains. It would be useful to conduct a study to determine how else the dynamic programming algorithm can be modified to reduce the bitwidth require-

[3]The theoretical maximum clock frequency is 550MHz; determining what frequency it is possible to achieve in practice requires trial-and-error with the design and the build tools.

[4]The idea presented in this section arose from discussions with Alex Krasnov, in which he posed the question, "But why can't you just pipeline it?".

ments, but still yield biologically relevant results, and then implement the optimal algorithm.

We have demonstrated that sequence alignment for resequencing maps very well to FPGA architectures. While our work is not yet appropriate for production use, we expect that with the further work we suggest, a very compelling system can be developed for broad use by the genomics community.

# Bibliography

[1] J. Venter, et al. The Sequence of the Human Genome. *Science*, **291**, 5507, 1304–1351 (2001).

[2] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, **409**, 860–921 (2001).

[3] Illumina Genome Analyzer Pipeline. URL: http://www.illumina.org.

[4] 1000 Genomes: A Deep Catalog of Human Genetic Variation. URL: http://www.1000genomes.org.

[5] Drosophila Population Genomics Project. URL: http://www.DPGP.org.

[6] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman. Basic Local Alignment Search Tool. *J. Mol. Biol.*, **215**, 3, 403–410 (1990).

[7] S. Batzoglou, D. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. Mesirov and E. Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Res.*, 12(1), 177–189 (2002).

[8] J. Mullikin and Z. Ning. The Phusion Assembler. *Genome Res.*, 13(1), 81–90 (2003).

[9] A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner and S. Batzoglou. Whole-genome sequencing and assembly with high-throughput short-read technologies. *PLOS One*, 2(5), e484 (2007).

[10] M. Pop and S. Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, **24**, 3, 142–149 (2008).

[11] F. De Bona, S. Ossowski, K. Schneeberger and G. Rtsch. Optimal spliced alignments of short sequence reads. *Bioinformatics*, 24(16), 174–180 (2008).

[12] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, **34**, 2, 171–210 (2002).

[13] M. Gokhale and B. Schott. Data-parallel C on a reconfigurable logic array. *J. Supercomput.*, **9**, 3, 291–313 (1995).

[14] C. Chang, J. Wawrzynek and R. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Comput.*, **22**, 2, 114–125 (2005).

[15] P.-Y. Droz. *Physical Design and Implementation of BEE2: A High End Reconfigurable Computer.* M.S. Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (2005).

[16] C. Chang. *Design and Applications of a Reconfigurable Computing System for High Performance Digital Signal Processing.* Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (2005).

[17] H. So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers.* Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley (2007).

[18] H. So and R. Brodersen. A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH. *ACM Transactions on Embedded Computing Systems*, **7**, 2 (2008).

[19] Xilinx History. URL: http://www.xilinx.com/company/history.htm

[20] J. Hennessy and D. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA. (2006).

[21] S. Hauck and A. DeHon. *The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann, Burlington, MA. (2007).

[22] G. Estrin. Organization of Computer Systems–The Fixed Plus Variable Structure Computer. In *Proc. Western Joint Computer Conf.*, 1960, pp. 33 – 40, New York, NY.

[23] G. Estrin, B. Bussel, T. Turn and J. Bibb. Parallel processing in a reconstructable computing system. *IEEE Trans. Elect. Comput.*, 1963, pp. 747 – 755.

[24] K. Asanovic, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *UC Berkeley Technical Report*, 2006, UCB/EECS-2006-183.

[25] K. Asanovic, et al. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. *UC Berkeley Technical Report*, 2008, UCB/EECS-2008-23.

[26] Stanford Pervasive Parallelism Laboratory. URL: http://ppl.stanford.edu.

[27] Xilinx Virtex 5 Product Table. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/v5product_table.pdf.

[28] Intel Microprocessor Export Compliance Metrics. URL: http://www.intel.com/support/processors/sb/CS-023143.htm.

[29] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman, Boston, MA. (1995).

[30] *Xilinx Virtex 5 User Guide.* UG190, Xilinx. (2008).

[31] G. Zhang, et al. Reconigurable Acceleration for Monte Carlo based Financial Simulation. In *Proc. Intl Conf. on Field Prog. Tech.*, 215–222. (2005).

[32] R. Karanam, A. Ravindran, A. Mukherjee, C. Gibas and A. Wilkinson. Using FPGA-Based Hybrid Computers for Bioinformatics Applications. Xilinx Xcell Journal, **58**, 80-83 (2006).

[33] C. He, M. Lu and C. Sun. Accelerating seismic migration using FPGA-based coprocessor platform. In *Proc. 12th IEEE Symp. on Field-Prog. Cust. Comput. Machines*, 207–216 (2004).

[34] A. Parsons, D. Backer, C. Chang, et al. PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy. *Proc. 10th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, November 2006.

[35] N. Asadi, T. Meng and W. Wong. Reconfigurable computing for learning Bayesian networks. In *Proc. 16th Intl. Symp. on Field Prog. Gate Arrays*, 203–211 (2008).

[36] S. Needleman and C. Wunsch. 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).

[37] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, **147**, 195–197 (1981).

[38] V. Kindratenko, D. Pointer, D. Raila and C. Steffen. Comparing CPU and FPGA Application Performance. *ISL White Paper, National Center for Supercomputing Applications* (2006).

[39] R. Lipton and D. Lopresti. A systolic array for rapid string comparison. In *Proc. Chapel Hill Conference on Very Large Scale Integration*, 363-376 (1985).

[40] D. Hoang. *A systolic array for the sequence alignment problem.* Technical Report, Department of Computer Science, Brown University, (1992).

[41] P. McMahon. *High Performance Reconfigurable Computing for Science and Engineering Applications.* Undergraduate Thesis, Department of Electrical Engineering, University of Cape Town (2006).

[42] P. McMahon, K. Stevens, H. Chen, T. Filiba, V. Nagpal and Y. Song. Parallel alignment of multiple short-read sequences against a reference genome on a reconfigurable computer cluster. *Submitted.* (2008).

[43] G. Estrin, B. Bussel, T. Turn and J. Bibb. Parallel processing in a reconstructable computer system. *IEEE Trans. Elect. Comput.*, 747–755 (1963).

[44] C. Chang, K. Kuusilinna, B. Richards and R. Brodersen. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Proc. Eleventh ACM Intl. Symp. on FPGAs*, 91–99 (2003).

[45] M. Gokhale, et al. SPLASH: A Reconfigurable Linear Logic Array. In *Proc. Intl. Conf. Parall. Proces.*, 526–532 (1990).

[46] J. Arnold, D. Buell and E. Davis. Splash 2. In *Proc. Fourth Annual ACM Symp. on Parall. Algor. Arch.*, 316–322 (1992).

[47] SRC Computers. URL: http://www.srccomp.com.

[48] VHDL Language Reference Manual. IEEE Standard 1076-2002.

[49] Verilog Hardware Description Language. IEEE Standard 1364-2001.

[50] D. Luebke, et al. GPGPU: general-purpose computation on graphics hardware. In *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, 208 (2006).

[51] J. Kahle, et al. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, **49** 4, 589–604 (2005).

[52] A. Grama, G. Karypis, V. Kumar and A. Gupta. *Introduction to Parallel Computing.* Pearson Education, Essex, England. (2003).

[53] N. Petkov. *Systolic Parallel Processing.* North-Holland, Amsterdam, The Netherlands. (1993).

[54] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, **23**, 156–161 (2007).

[55] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System In FPGAs. In *Proc. Intl Conf. Field Programmable Logic and Applications*, 54 – 61 (2007).

[56] Y. Yamaguchi, T. Maruyama and A. Konagaya. High speed homology search with FPGAs. In *Proc. Pacific Symposium on Biocomputing*, 271–282 (2002).

[57] P. Zhang, G. Tan and G. Gao. Implementation of the Smith-Waterman Algorithm on A Reconfigurable Supercomputing Platform. In *Proc. First Intl Workshop on High-Performance Reconfigurable Computing Technology and Applications*, 39–48 (2007).

[58] L. Hasan, Z. Al-Ars and S. Vassiliadis. Hardware Acceleration of Sequence Alignment Algorithms — An Overview. In *Proc. Design and Technology of Integrated Systems in Nanoscale Era*, 92–97 (2007).

[59] L. Pachter and B. Sturmfels. (Eds.) *Algebraic Statistics for Computational Biology.* Cambridge University Press, Cambridge, England. (2005).

[60] R. Batista, A. Boukerche and A. de Melo. A Parallel Strategy for Biological Sequence Alignment in Restricted Memory Space. *J. Parallel and Distributed Computing*, **68**, 548–561 (2007).

[61] J. Shendure, R. Mitra, C. Varma and G. Church. Advanced Sequencing Techniques: Methods and Goals. *Nat. Rev. Genet.*, 5(5), 335–344 (2004).