Enabling the Use of Heterogeneous Computing for Bioinformatics.

Ramakrishna Bijanapalli Chakri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

September 16, 2013
Blacksburg, Virginia

Keywords: FPGA, Hardware Acceleration, High Performance Computing, DNA Alignment, LabVIEW, Heterogeneous Computing, GP-GPUs.

Enabling the Use of Heterogeneous Computing for Bioinformatics

Ramakrishna Bijanapalli Chakri

(ABSTRACT)

The huge amount of information in the encoded sequence of DNA and increasing interest in uncovering new discoveries has spurred interest in accelerating the DNA sequencing and alignment processes. The use of heterogeneous systems, that use different types of computational units, has seen a new light in high performance computing in recent years; However expertise in multiple domains and skills required to program these systems is causing an hindrance to bioinformaticians in rapidly deploying their applications into these heterogeneous systems. This work attempts to make an heterogeneous system, Convey HC-1, with an x86-based host processor and FPGA-based co-processor, accessible to bioinformaticians. First, a highly efficient dynamic programming based Smith-Waterman kernel is implemented in hardware, which is able to achieve a peak throughput of 307.2 Giga Cell Updates per Second (GCUPS) on Convey HC-1. A dynamic programming accelerator interface is provided to any application that uses Smith-Waterman. This implementation is also extended to General Purpose Graphics Processing Units (GP-GPUs), which achieved a peak throughput of 9.89 GCUPS on NVIDIA GTX580 GPU. Second, a well known graphical programming tool, LabVIEW is enabled as a programming tool for the Convey HC-1. A connection is established between the graphical interface and the Convey HC-1 to control and monitor the application running on the FPGA-based co-processor.

This work was supported in part by the I/UCRC Program of the National Science Foundation (NSF) within the NSF Center for High-Performance Reconfigurable Computing (CHREC), Grant No. IIP-0804155.

Dedication

I dedicate this work to my parents and to my uncle & aunt who brought me up.

Acknowledgments

This thesis would not have been possible without the guidance and help of several individuals. I would like to take this opportunity to acknowledge them.

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Peter Athanas, for giving me the opportunity to be a part of the Configurable Computing Lab (CCM). He has been an immense source of inspiration to me throughout the process. I am grateful for his patience and his continued support of me at every stage of my academic life at Virginia Tech. It would have been great to pursue a course work taught by him, which I unfortunately could not take.

Dr. Alan Dickerman and Dr. Wu-chun Feng, for graciously agreeing to serve on my thesis committee on a short notice. I also had the privilege of taking the computer architecture course taught by Dr Feng, which was inspiring and provided me with a great learning experience.

Dr. Krzystof Kepa, for his valuable insights and for helping me complete my work on time. It was great to meet a person with strong dedication and amazing wit like him.

David Uliana, for all his help over the two years, for inviting me to his home for a thanksgiving dinner and introducing me to his warm and generous family members.

Shaver Deyerle, for all the fun we had together and for the insightful interactions on several academic and non academic matters.

Kavya Shagrithaya, for introducing me to this lab, for helping me choose my course work and for guiding me during my admission process.

My other lab mates: Ali, for all the endless conversations about the world of technology, Tannous, for all the advice and technical support in using my lab computer, Andrew, for including me in his meal plan at D2 over the summer and for sharing several interesting facts on daily basis, Ritesh, for being supportive as a batch-mate and giving me company and support on various issues over the two years, Kaiwan, for his generous help and for being very approachable anytime I seek information from him, Kevin, for his support during the project and all the valuable interactions we had. My former lab members Abhay and Umang, for their support and guidance.

All my CCM lab mates for all the endless fun we had with bike rides, Foosball tournaments, picnics and lunch at El Rodeo and Happy Wok. Members of Wearable lab for their support on various occasions, particularly Jacob for driving his pickup truck to help me move.

The administrative staff of the Graduate School and the Bradley Department of Electrical and Computer Engineering for their timely help and cooperation.

My amazing friends at Virginia Tech, Apoorva, Aruna, Dilip, Harish, Rashmi, Shuchi, Supratik and many others, for their support in every aspect of my academic life here. My numerous other friends in various parts of the world for extending their help from far.

My parents, my uncle and aunt, my siblings and my cousins for their unconditional love and support during my graduate school life.

Last but not the least, God for showering His blessings on me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Genomics is one of the emerging fields that offers a huge challenge to both biologists and computer scientists due to the magnitude of data involved in a genome. For instance, the human genome has three billion nucleotide base pairs, and it holds a great deal of information. This huge amount of data requires constant processing for researchers all over the world to make new discoveries in the field of genomics. These computationally intensive problems take up a huge amount of time to execute on a serial computer. Programmers all over the world are trying to find new ways to accelerate this process, either by formulating new algorithms or by deploying them on unconventional systems and custom hardware to solve these problems.

Heterogeneous computing systems are an example for these unconventional systems. These systems use a variety of different types of computational units like a Graphics Processing Unit (GPU), a co-processor, or a Field-Programmable Gate Array (FPGA) along with a Central Processing Unit (CPU). Unfortunately, these heterogeneous systems bring in new and unexplored territories to bioinformaticians and programmers, making it arduous for them to rapidly deploy applications on these heterogeneous platforms.

1.1 Motivation

This work is motivated by the fact that the heterogeneous systems based on FPGAs with a high potential for reconfigurability and application acceleration has been out of reach of bioinformaticians. Programming the accelerator requires training in digital logic design, expertise in HDL programming and a working knowledge of heterogeneous architectures including several interfaces that are specific to these machines.

Some of the prior work involving the use of heterogeneous architectures for bioinformatics have been made primarily for benchmarking the performance of the machine and is of little use for the bioinformaticians. The hardware design engineer lacks the skills to decipher various biological requirements and parameters, which brings the need for bioinformaticians to use the low level hardware interfaces from the accelerator in their applications.

The Virginia Bioinformatics Institute (VBI) owns several nodes of FPGA-based heterogeneous computers from Convey Computers called HC-1 and HC-1ex. Convey demonstrated a huge performance gain with a dynamic programming based alignment algorithm called Smith-Waterman, on their HC-1ex platform [3]. This work involves the development of accelerator for Smith-Waterman. The following are the reasons for designing a custom version of dynamic programming personality.

1. Dynamic programming kernels can be mapped efficiently in hardware by employing systolic arrays.
2. A custom design can be altered for one's own benefits.
3. This can be used as a template to implement any dynamic programming solution by modifying the source thats already there. Ex Needleman-Wunch [4] , CLUSTAL, GeneScan.
4. Applications can be created in the host that make use of dynamic programming in hardware. Algorithms such as BWASW [5], Swift [6] and BFAST [7] directly use Smith-Waterman. There could be several other such algorithms that benefit from an accelerator for dynamic programming.

Need for an Unconventional Approach
Use of conventional approach to programming heterogeneous systems involves skills in multiple programming paradigms and is complicated by different platforms and different programming languages. The components of the heterogeneous systems often involve sub-systems integrated from different vendors. These often lead to difficulty in timing, synchronization and handshake protocols. All of these problems inherent in these system provide a hindrance to bio-scientists from using them. Thus, providing a intuitive and unified solution to program the heterogeneous systems enables the bioinformaticians to develop applications for these systems.

1.2 Contributions

In this work, a two-fold approach to solve the problem is proposed. One approach is to use heterogeneous architectures for bioinformatics. Several levels of possible optimization were made to achieve highest performance achievable. The other approach is to use unconventional methods to make this huge computation power available to computational biologists.

For the first task, a highly efficient dynamic programming kernel for Smith-Waterman is built. Interfaces are defined and tested for its functionality. This interface is made available to any application that requires dynamic programming. The performance is compared with a serial computer as well as with a similar implementation on General Purpose-Graphics Processing Units (GP-GPUs).

For the second task, a versatile and mature graphical programming language is chosen. It is made to work with an heterogeneous environment and tested for the functionality. A communication channel is formed between the Graphical User Interface (GUI) based front-end and a back-end design running on the heterogeneous system. This is finally tested from end-to-end with hardware simulation as well as the run-time execution.

1.3 Organization

The crux of this thesis consists of four major chapters which is organized as follows,

Chapter 2 - Background: This chapter familiarizes the concepts behind DNA sequencing and alignments, FPGA architecture, programming and its tools, Convey architecture and its personality development environment, LabVIEW and its FPGA programming capabilities, and finally the General Purpose Graphics Processing Units and its programming paradigm, CUDA.

This chapter also highlights some of the previous work in the field of hardware based accelerations for bioinformatics applications, GP-GPU based Smith-Waterman implementations and graphical and high level programming models for hardware.

Chapter 3 - High Performance Smith-Waterman Implementation: The implementation of first part of the contribution along with the approach taken, is provided in this chapter. This chapter details the implementation of the Smith-Waterman cell to the final integration with the Convey memory. This chapter ends with a description of the implementation of the Smith-Waterman algorithm on GP-GPUs.

Chapter 4 - Convey LabVIEW Interface: The implementation of the second part of the contribution along with the approach taken is provided in this chapter. This chapter contains elaborate details on the LabVIEW generated HDL, its interface to the Convey environment and finally its connections to the LabVIEW VI front panel through specific interfaces.

Chapter 5 - Results and Analysis: This chapter exhibits the results obtained, and the analysis of those results. The performance of the FPGA based Smith-Waterman is compared to that of the GP-GPU. Finally it compares this work with some of the previous work done in this area.

Chapter 2

Background

2.1 DNA Sequencing

DNA carries the genetic information in the cell and is capable of self-replication and synthesis of RNA. DNA consists of two long chains of nucleotides twisted into a double helix and joined by hydrogen bonds between the complementary bases adenine (A) and thymine (T) or cytosine (C) and guanine (G). In order to read and analyze any genetic sequence, biologists must first determine a sample DNA sequence by reading the sequence of nucleotide bases and must compare them to a reference or a known genome. Since the human DNA is a three billion base pair sequence, this becomes a difficult computational problem.

To determine the sequence of base pairs, the DNA sample is first replicated to produce approximately thirty copies of the original DNA sequence. The copies of the replicated sequence are then cut at random points throughout the entire length of the genome, producing short lengths of intact chains of base pairs known as short reads [8]. DNA sequencing technology began with Sanger sequencing in 1977 and evolved to many new massively parallel sequencing technologies such as Illumina. The Illumina/Solexa sequencing technology typically produces 50 to 200 million 32 to 100 base pair reads on a single run of the machine [9]. However, other new sequencing techniques such as Roche/454 sequencing technology has produced long reads >400 base pair in production, Illumina is gradually increasing read length >100 base pair, and Pacific Bioscience generates 1000 base pair reads in early testing [5] requiring a need for both short and long read alignments.

These massively parallel sequencing technologies produce a large volume of short reads, that mapping all these short reads to a large genome presents a great demand to develop faster sequence alignment programs.

2.2 Sequence Alignment

Sequence alignment is defined as a process of arranging the sequences of DNA, RNA, or a protein against a known reference to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [10]. It is the process of mapping the reads obtained from one of the DNA sequencing techniques to a known reference. It attempts to map reads to the reference genome in order to study the newly sequenced DNA segment or reconstruct an entire DNA.

This process of read mapping involve huge quantities of data, demanding a large amount of memory and computational power. This often leads to long execution times in standard processors. Several algorithms have been developed to solve sequential alignment problems. Dynamic programming based algorithms such as Smith-Waterman and Needleman-Wunch, described in Section 2.2.3, provide optimal alignments in quadratic time on a serial processor [11]. Nonetheless, there have been methods proposed to reduce the computation time by sacrificing some accuracy of the solution leading to sub-optimal alignments. Most of these algorithms filter specific locations in reference genome to perform alignment instead of looking for alignments over the stretch of entire genome, which is both time consuming and redundant. Most algorithms can be categorized to accomplish this either from a type of indexing or by Burrows Wheeler Transform (BWT).

2.2.1 Indexing Based Approach

The indexing based solutions attempt to find subsequences of each read that match perfectly to a location in the reference genome. In this method, an index of the reference genome is created, which maps every short subsequence, called seeds, that occurs in the reference genome to the locations where they occur. To align a short read, all of the seeds in the read are looked up in the index, which yields a set of Candidate Alignment Locations. These seeds should perfectly match to at least one location in the reference genome. Because the seeds will match perfectly to the reference, an index of the reference can be created for the desired seed length that tells at which positions in the reference a given seed occurs. These seeds are then extended to form a full alignment of that read. The location with the highest score is chosen as the alignment location for a short read. BFAST [7] is an example of a program based upon this algorithmic approach.

2.2.2 Burrows Wheeler Transform Based Approach

The second category of algorithm that attempts to solve the short read mapping problem, uses a data compression structure called the Burrows-Wheeler Transform (BWT) [12] and an indexing scheme known as the FM-index [13]. The core idea for searching the genome to find the exact alignment for a read is rooted in the suffix trie theory [14]. The BWT of a sequence of characters is constructed by creating a square matrix containing every possible rotation of the desired string, with one rotated instance of the string in each row. The matrix is then sorted lexicographically, and the BWT is the sequence of characters in the last column, starting from the top.

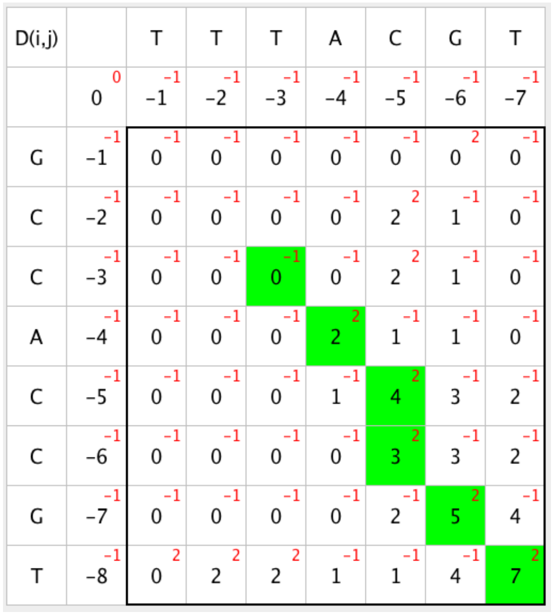
This solution uses the FM-index, a data structure that synergistically combines the Burrows-Wheeler transform and the suffix array, to efficiently store information required to traverse a suffix tree for a reference sequence. These solutions can quickly find a set of matching locations in a reference genome for short reads that match the reference genome with a limited number of differences. However, the running time of this class of algorithm is exponential with respect to the allowed number of differences; therefore BWT-based algorithms tend to be less sensitive than others. Bowtie and BWA [9] are examples of programs based upon this algorithmic approach.

2.2.3 Dynamic Programming and Smith-Waterman

The problem of sequence alignment can be characterized to be solved by dynamic programming algorithms where the bigger problem is broken into smaller similar problems. Because of the similar nature of the smaller problems, one can take advantage of a highly data parallel hardware designs. It is especially beneficial when a large amount of data has to be processed while dealing with long sequences or a large number of smaller sequences.

Dynamic programming is used in many areas of computing and finds a great application in finding best alignments in genome sequences. The disadvantages of DNA alignments using dynamic programming are quadratic time and space complexities [15]. The use of parallel execution can reduce the time complexity to linear time. One solution is to use a highly data parallel processor like a GPU. Another solution is to design a specific processing kernel in hardware and replicate it several times using a FPGA. These solutions can achieve massive amounts of parallelism. Hence FPGAs and GPUs form an excellent platform for genome alignment using dynamic programming.

Smith-Waterman [16], based on dynamic programming, is the one of the popular algorithms for finding local alignments given a query and a reference sequence. It calculates the score matrix for two given sequences using the *equations* 2.1a to 2.1d that describe the working of the algorithm. It involves the use of a score matrix that keeps the track of the scores as the computations are being done. The alignments are performed by tracing the path from the highest score in the matrix to a point where the score drops to zero. This is termed as the *trace-back*. Figure 2.1 shows the alignment of two sequences $TTTACGT$ and $GCCACCGT$ with the score matrix for the alignment and its trace-back. Based on the score matrix, the threshold point is chosen from where the trace-back starts to find all the possible optimal



D(i,j)		T	T	T	A	C	G	T
	0^{0}	-1^{-1}	-2^{-1}	-3^{-1}	-4^{-1}	-5^{-1}	-6^{-1}	-7^{-1}
G	-1^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{2}	0^{-1}
C	-2^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{-1}	2^{2}	1^{-1}	0^{-1}
C	-3^{-1}	0^{-1}	0^{-1}	🟩 0^{-1}	0^{-1}	2^{2}	1^{-1}	0^{-1}
A	-4^{-1}	0^{-1}	0^{-1}	0^{-1}	🟩 2^{2}	1^{-1}	1^{-1}	0^{-1}
C	-5^{-1}	0^{-1}	0^{-1}	0^{-1}	1^{-1}	🟩 4^{2}	3^{-1}	2^{-1}
C	-6^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{-1}	🟩 3^{2}	3^{-1}	2^{-1}
G	-7^{-1}	0^{-1}	0^{-1}	0^{-1}	0^{-1}	2^{-1}	🟩 5^{2}	4^{-1}
T	-8^{-1}	0^{2}	2^{2}	2^{2}	1^{-1}	1^{-1}	4^{-1}	🟩 7^{2}

Figure 2.1: Smith-Waterman Alignment Score Matrix.

alignments.

$$H(i,0) = 0, 0 \leq i \leq m \ldots\ldots \tag{2.1a}$$

$$H(0,j) = 0, 0 \leq j \leq n \ldots\ldots \tag{2.1b}$$

$$if\ \ a_i = b_j\ then\ w(a_i, b_j) = w(match)\ or\ if\ \ a_i \neq b_j\ then\ w(a_i, b_j) = w(mismatch) \tag{2.1c}$$

$$H(i,j) = max \begin{cases} 0 & , \\ H(i-1, j-1) + w(a_i, b_j) & Match/Mismatch, \\ H(i-1, j) + w(a_i, -) & Deletion, \\ H(i, j-1) + w(-, b_j) & Insertion. \end{cases} \tag{2.1d}$$

Needless to mention, dealing with huge genome sequences is a highly computation intensive task. With the advent of parallel computing technologies in general and massively data parallel architectures like FPGAs and GP-GPUs in particular, there has been a breakthrough in solving the above mentioned alignment problem in reasonable time without sacrificing the accuracy.

2.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays or FPGAs in short are the class of semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. This hardware can be reconfigured for specific applications even after the product has been installed in the field, hence the name ”field-programmable” [17]. FPGAs can be used to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

Xilinx and Altera are the largest producers of FPGA at the time of writing. FPGA are the crux of the user programmable accelerators which is the characteristic feature of the Convey HC-1, the platform that is being used in this work.

2.3.1 FPGA Architecture

FPGAs have a uniform layout of logic blocks, input/output blocks and the interconnects between them. Xilinx and Altera FPGAs have the same basic architecture but they use different sets of tools and terminologies. As Xilinx FPGAs are used in Convey Computers, the components and tools are defined in the context of Xilinx FPGAs.

Configurable Logic Block (CLB), the basic logic elements for Xilinx FPGAs, provide combinatorial and synchronous logic as well as distributed memory and SRL32 shift register capability [18]. A single CLB in Virtex-5 family of FPGAs consists of two slices: SLICEL (logic) and SLICEM (memory). Each CLB is connected to a switch matrix which can access a general routing (global) matrix. Every slice contains four Look-Up Tables (LUTs), wide function MUXs, carry logic, and configurable memory elements. SLICEM support storing data using distributed RAM and data shifting with 32-bit shift registers. Look-up tables, implement the entries of a logic functions truth table. LUTs are used for combinational functions or can also be used to implement small Random Access Memories (RAM). Flip Flops (FFs)/Latches are the memory elements with programmable clock edges, set/reset and clock enable. These memory elements can also be configured as shift registers.

I/O blocks (IOBs) provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by programmable I/O blocks (IOBs). The IOBs can be connected to a flexible ChipSync logic for enhanced source-synchronous interfacing. Source-synchronous optimizations include per-bit deskew (on both input and output signals), data serializers/deserializers, clock dividers, and dedicated I/O and local clocking resources. The Input Output Buffers (IOB) are programmable for inputs or outputs at the periphery of the FPGA chip. The I/O voltages and currents are also configurable.

Block RAM modules provide flexible 36 Kbit true dual-port RAM that are cascadable to form larger memory blocks. In addition, Virtex-5 FPGA block RAMs contain optional programmable FIFO logic for increased device utilization. Each block RAM can also be configured as two independent 18 Kbit true dual-port RAM blocks, providing memory granularity for designs needing smaller RAM blocks.

Programmable Interconnect Network contains horizontal and vertical mesh of wire segments interconnected by programmable switches called programmable interconnect points (PIPs). These PIPs are implemented using a transmission gate controlled by a memory bits from the configuration memory.

2.3.2 FPGA Tools

Described here are the tools that are used during the various stages of an application development on FPGAs.

Simulation Tools are either the tools provided by the FPGA vendors or third party companies that help in testing the functionality of the design before being downloaded on to the FPGA. This helps in saving development time between iterations and also aids in encapsulating the logic flow in the design. For Convey designs, Synopses VCS is used for simulation.

Waveform Analyzers are the tools that gather the dump files from the simulation tools and draw the timing diagrams for the programmer to analyze the flow of data and signals between the various design components. Convey supports Synopsys DVE tool to analyze the waveforms.

Synthesis tools are most often provided by the FPGA vendors and is used for the entire design flow from compilation of the source code, generation of the schematic from the source, logic placement, routing to generation of the bit-stream that finally programs the FPGA. In this work, Xilinx ISE 14.1 is used for all the synthesis purposes.

Debug tools, are used alike probes to debug the design on the actual hardware. Several probing points are incorporated at the time of design and these tools lets the designers look into the logic fabric to debug hangs and incorrect functionalities in the design. Chipscope, a Xilinx tool, is used to debug issues on Convey HC-1 hardware.

2.3.3 Programming FPGAs

Traditionally, FPGAs were programmed using Hardware Descriptions Languages (HDLs), but recently several high level techniques were introduced to for programming FPGAs. Described below are some of the techniques available today, to program FPGAs.

Hardware Description Languages are the traditional and most often used design language for FPGA. VHDL and Verilog are two popularly used Hardware description languages. The use of HDLs allow the designer to control every minute detail in the design and how its being configured by a resource on FPGA. However programming in HDL requires the knowledge of hardware and Digital design, which is not accessible to all.

High Level Synthesis is a designing approach where the programmer develops hardware in high level language such as C. Xilinx AutoESL is a tool developed by Xilinx to accomplish this. Although this in not used in this work, it is provided here for completion.

OpenCL is similar to AutoESL with additional support to the OpenCL libraries. This is still an area of research and no commercial tool is available at the time of writing.

CORE Generator is another high level design tool from Xilinx, primarily developed to generate efficient cores for some regularly used complex functions such as a floating point computation.

Azido is one of the graphical entry tools developed by a 3^{rd} party vendor, Data IO. This was the tool used in the design of Bflow [19], which is used as reference for LabVIEW based ConVI in this work.

LabVIEW is another graphical programming environment developed by National Instruments primarily for their data acquisition devices. However NI extends its LabVIEW support to various FPGA boards. This work involved use of LabVIEW for designing the accelerators in Convey.

2.4 Convey HC-1

The Convey HC-1 is hybrid-core computer, one that improves application performance by combining an x86 processor with hardware that implements application-specific instructions. It uses a commodity two-socket motherboard to combine a reconfigurable, FPGA-based co-processor with an industry standard Intel Xeon serving as the host processor. Convey uses a mezzanine connector to extend the Front Side Bus (FSB) interface to a large co-processor board. The architecture is shown in Figure 2.2. Convey HC-1 combines Intel Xeons CPU and four Xilinx Virtex 5 FPGAs in its Application Engines (AE) [2].

2.4.1 Host Processor

The *Host*, an Intel Xeon processor executes standard x86 applications in a Linux environment. The host application communicates to the hardware accelerator by passing parameters to the Convey assembly code which are in turn directed to the application engine through the scalar processor and the dispatch interface.

2.4.2 Application Engine Hub (AEH)

The *Application Engine Hub* is made of two non-user programmable FPGAs, one that interfaces to the host processor through the FSB and the other containing the scalar soft core

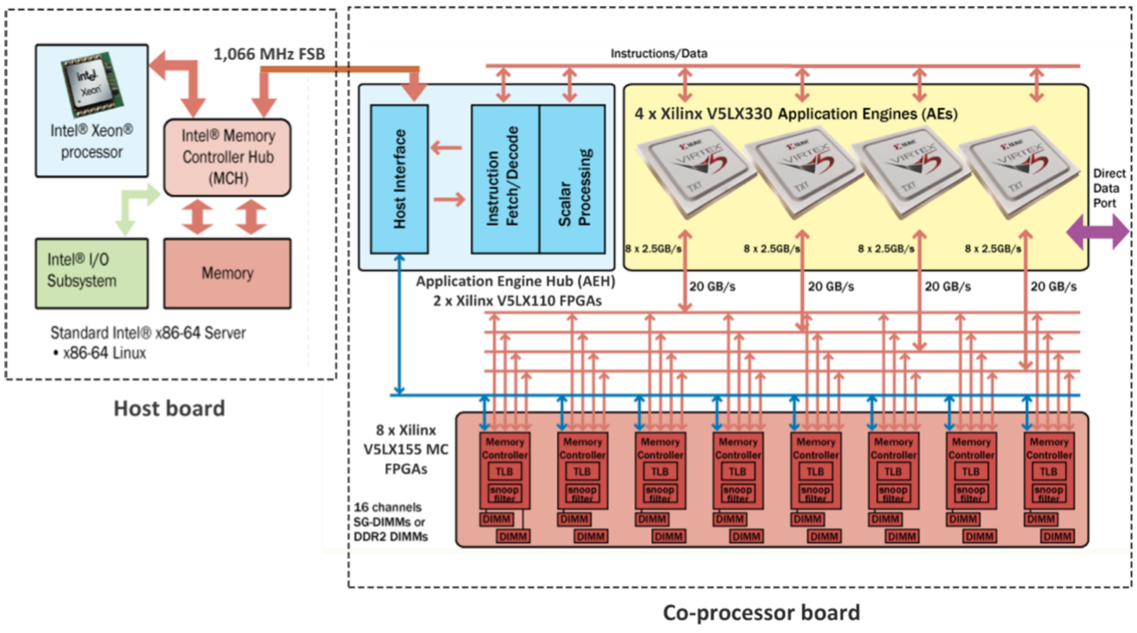


Figure 2.2: Convey Architecture. Source: [1]. Used under fair use guidelines, 2011.

Table 2.1: Convey HC-1 Specifications

Parameter	HC-1	HC-1ex
Clock Speed	2.13 Ghz Dual Core	2.13 Ghz Quad Core
Host Memory	4-16 DIMM max 128 GB	4-16 DIMM max 128 GB
Co-Processor Memory	8-16 DIMM max 128	8-16 DIMM max 128
AE-AE	668 MBps full duplex ring	668 MBps full duplex ring
Co-Processor (FPGAs)	4*Virtex 5 (XC5VLX330)	4*Virtex 6 (XC6VLX760)
AE-Memory	2.5 GBps link	2.5 GBps link
SG DIMM	5 GBps	5 GBps
Peak Bandwidth	20 GBps with striding	20 GBps with striding
Co-Processor Memory	16 GB	16 GB
Co-Processor Clock Speed	150 MHz	150 MHz

and its interface to the application engines called the dispatch interface. The scalar processor runs the Convey assembly code and executes the instructions defined in the assembly language. It then sends data and signals decoded from the instructions to the application engine FPGAs though the dispatch interface as shown in Figure 2.3.

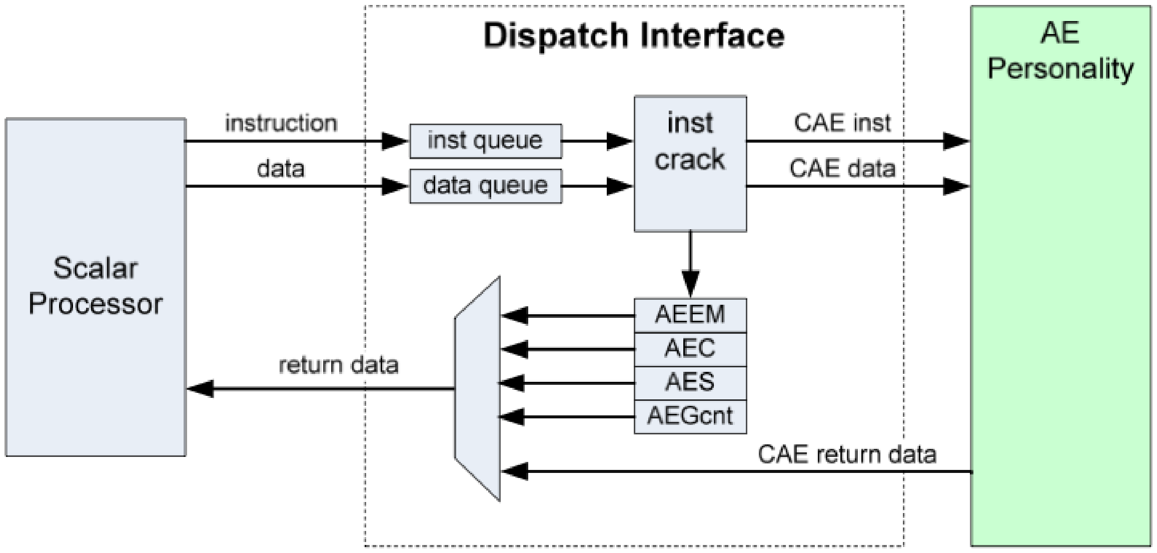


Figure 2.3: Convey AE Hub and the Dispatch Interface. Source: [2]. Used under fair use guidelines, 2011.

2.4.3 Application Engines (AEs)

Four user programmable FPGAs that constitute the compute core for the co-processor are called the *Application Engines*. Detailed description of the Convey application engines are provided in Table 2.2 for the HC-1 and HC-1ex systems. The FPGA used here are the Xilinx Virtex family of high performance FPGAs [18]. The AEs themselves are interconnected in a ring configuration with 668 MB/s, full duplex links for AE-to-AE communication.

2.4.4 Dispatch Interface

The host processor communicates with the AEs through the *Dispatch Interface*. The components of the dispatch interface are shown in Figure 2.3. In order to communicate with the AEs, the host code uses API functions called *co-processor calls* or *copcalls*. The host CPU uses this mechanism to send parameters to the AEs and receive status information from the AEs [20]. Copcalls can either be a blocking copcall or a non-blocking copcall. In case of a

Table 2.2: The AEs Compared

Parameters	Virtex 5 (XC5VLX330)	Virtex 6 (XC6VLX760)
Slices	51840	118560
LUTs	Slices*4, 6 input	Slices*4
Flip Flops	Slices*4	Slices*8
Distributed RAM	3420Kb	8280Kb
DSP Slices	192	864
Block RAM	11664Kb	25920Kb
I/O Banks	27	30
Max I/O	960	1200

blocking call, the host application stalls until the co-processor returns control to the host application, whereas in case of a non-blocking copcall, the host application continuous to execute the instructions following the copcall and does not to wait for the co-processor to complete the execution. Non-blocking copcalls execute faster and are used in certain cases to avoid stalling the host processor.

2.4.5 Memory Interface

Convey co-processor contains eight memory controllers which are physically implemented in non user programmable FPGAs on the co-processor board. The four AEs interface with the eight memory controllers through a full crossbar. Each memory controller FPGA controls two DDR2 DIMMs. Each AE has a 2.5 GB/s link to a memory controller which in turn has a 5 GB/s link to its DIMMs. Each AE can achieve a theoretical peak bandwidth of 20 Gbyte/s when striding across eight different memory controllers. Figure 2.4 shows the memory connections in Convey [21].

2.4.6 Management Interface

Convey HC-1 includes an independent port to the application engines called the *management* or the *debug interface*. The management interface provides the communication path between the Management Processor (MP) [2] and the application engines. It was designed to be used for debugging and monitoring the status of the the FPGAs. Since this path is independent of the instruction dispatch path from the host processor, it can be useful in interfacing the LabVIEW VI by allowing access to the internal hardware design generated by LabVIEW without having to interrupt the host processor. This interface is made of registers called Control and Status Register (CSR) and these registers are grouped together to form a CSR

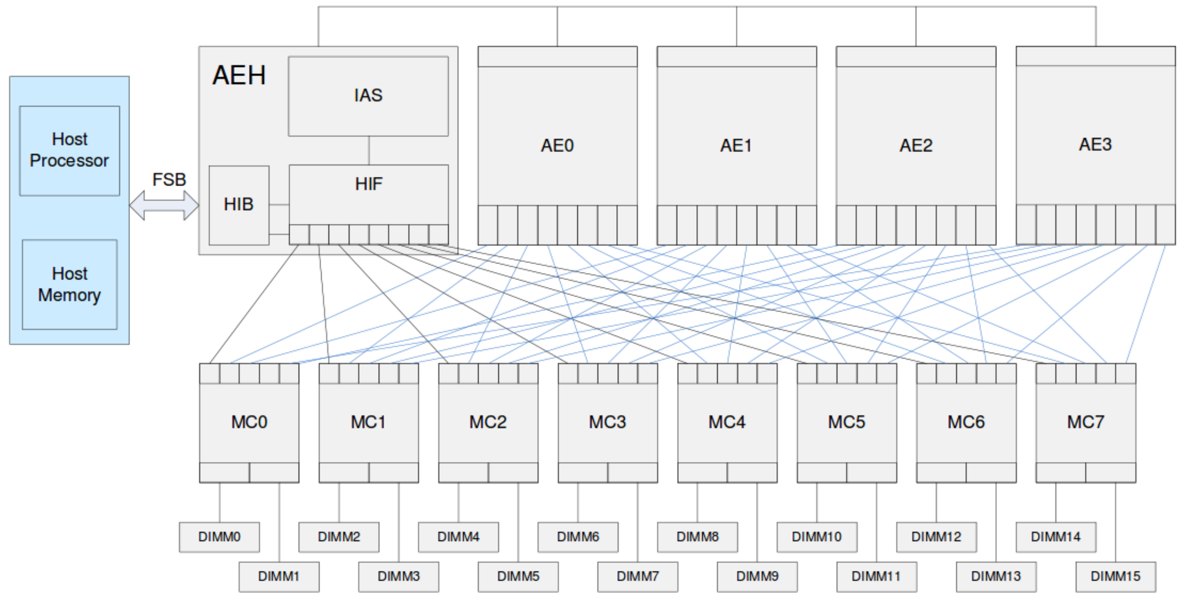


Figure 2.4: Convey Memory Connections. Source: [2]. Used under fair use guidelines, 2011.

agent. This interface is instantiated in the Convey supplied libraries, along with CSR agents in a ring topology as shown in Figure 2.5.

2.4.7 Personality

Personality is the Convey's term for the co-processor configuration consisting of hardware modules callable from the host by a copcall. Convey provides a personality development kit that enables the designer to develop custom personality for application engine. Convey provides logic libraries in the PDK to interface with the scalar processor, the memory controllers and the management processor. The developer develops the logic within this environment. The following are the Convey provided components as the part of the custom PDK.

1. Makefiles for both synthesis and simulation design flows.
2. Support for Verilog HDL. VHDL can be incorporated by adding commands to the existing Makefile. This comes to use while interfacing the LabVIEW modules.
3. Simulation modules for all the default interfaces provided by Convey.
4. A Programming-Language Interface (PLI) to let the host code interface with a behavioral HDL simulator such as Modelsim or Synopsys.

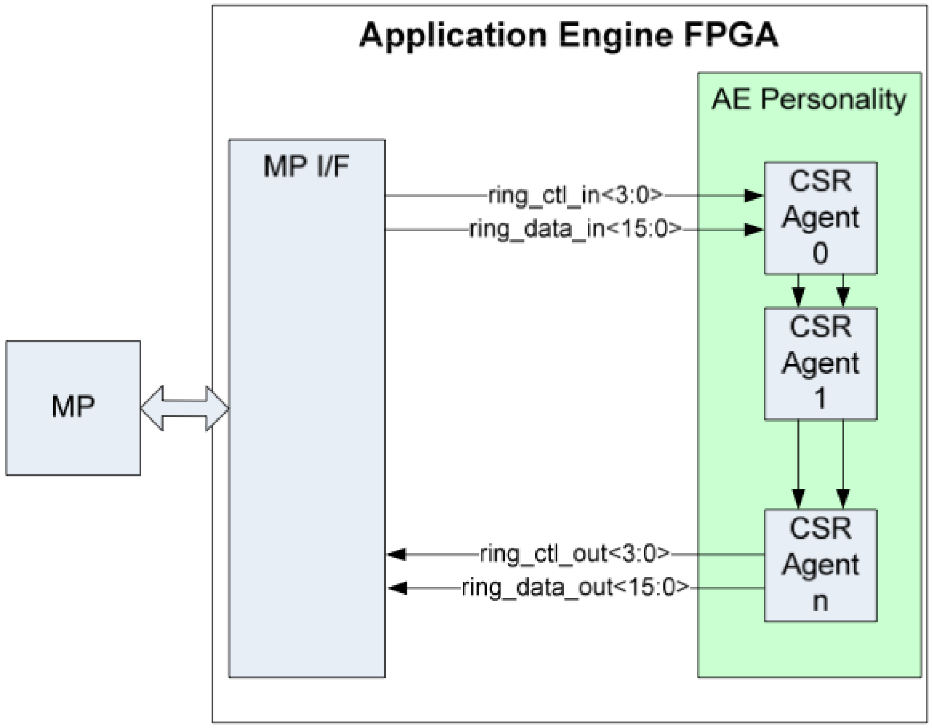


Figure 2.5: Convey Management interface with Signals. Source: [2]. Used under fair use guidelines, 2011.

2.5 LabVIEW FPGA

LabVIEW (short for Laboratory Virtual Instrument Engineering Workbench) is a graphical development platform from National Instruments. It has a front-end graphical interface called a Virtual Instrument (VI), which contains controls and indicators to interact with a program running in the back-end. The NI LabVIEW FPGA Module extends the LabVIEW visual programming language to target FPGAs on NI Reconfigurable I/O (RIO) hardware [22]. It was primarily created to configure the behavior of the reconfigurable FPGA to match the requirements of a specific measurement and control system. This specific VI created to run on an FPGA device is called the *FPGA VI* while the VI running on the host is called *Host VI*. In the remainder of the document, LabVIEW HDL (LV HDL) refers to the HDL (VHDL) generated from LabVIEW VI. The FPGA Module creates a register map, specific to the FPGA VI, that translates to a hardware register for every control and indicator used in VI. LabVIEW uses the register map internally to communicate with the FPGA VI from the interactive front panel, host VI, through the programmatic FPGA interface communication.

In this work, LabVIEW was chosen for the Convey's application engines design flow, and its called *ConVI*. Designing the accelerator with LabVIEW has its inherit advantages over conventional hardware design with HDL programming that can serve as a great benefit for the bioinformatics community for designing their own accelerators. The following are some of the benefits from using LabVIEW:

1. LabVIEW clearly represents parallelism in data flow, so users who are inexperienced in traditional FPGA design can productively apply the power of reconfigurable hardware.
2. No knowledge of HDL is required to design a specific hardware solution.
3. The user can design and rapidly develop hardware components with the power of LabVIEW graphical programming.
4. The user can use the Interactive Front Panel to communicate with the FPGAs from the host computer to control and test the algorithm running on the FPGA device.
5. LabVIEW also allows encapsulation of common sections of code as subVIs to facilitate their reuse on the block diagram.
6. The user can get to control and monitor data directly from the FPGA device using Interactive Front Panel Communication by reading and writing indicators and controls.
7. Indicators can be added to the FPGA VI block diagram to monitor the internal state of the FPGA VI. Indicators here serve the same purpose as probes for a non-HDL designer. An indicator can be placed anywhere on the block diagram where the user needs to see data to verify the functionality of the VI.

However, the use of LabVIEW for this work has some drawbacks. LabVIEW FPGA was geared towards embedded systems and not towards the heterogeneous systems like the Convey HC-1. There is also no prior work related to the use of LabVIEW to design the accelerators of a heterogeneous systems. Thus, these challenges had to be overcome before taking the advantage of using LabVIEW to design the accelerators for the Convey HC-1.

2.6 GP-GPUs and CUDA

Driven by the insatiable market demand for real time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and a high memory bandwidth. The GPU is especially well-suited to address problems that can be expressed as data-parallel computations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches [23].

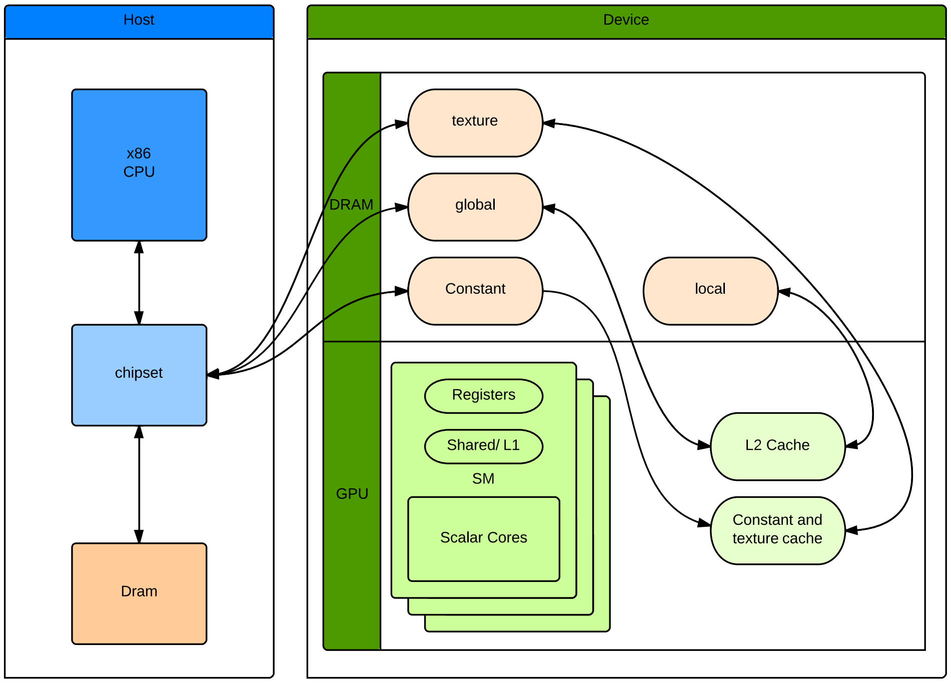


Figure 2.6: NVIDIA GP-GPU Architecture.

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient manner than on a CPU. The various techniques and the programming methods described in NVIDIA CUDA Programming Guide 2.0 [24] are used in this work to enable the use of NVIDIA General Purpose Graphics processing units (GP-GPUs) for this application.

2.6.1 Streaming Multiprocessors and CUDA Cores

Streaming Multiprocessors (SMs) are the physical execution units within the NVIDIA GP-GPUs. Each Streaming Multiprocessor has its own set of CUDA Cores (32 in Fermi Architecture), registers and shared memory which double up as the L1 caches for these cores. Each CUDA core is responsible for execution of a CUDA thread and each SM is responsible for scheduling and execution of entire CUDA thread block.

2.6.2 CUDA Memories

NVIDIA GP-GPUs have a hierarchy of different types of memories that serve the purpose of making GPUs efficiently execute data intensive application thereby increasing its throughput.

Figure 2.6 shows the relationship among various memories in a GPU.

Global memory is the Graphics-DDR (GDDR) DRAM residing outside the GPU on the Graphics card. These are made of GDDR memory as they have a wider data bus and higher clock frequency to crater to the higher bandwidth of data required by the computation cores of the GPU. Each access to the global memory takes approximately, 200 to 400 clock cycles, hence the latency is high while each access can provide 128 bytes of data increasing the throughput of access. Global memory is also the only memory accessible by the host CPU.

Local memory is a part of the GDDR DRAM that is dedicated to each of the streaming multiprocessor. These are not accessible by the host CPU and is generally cached.

Texture and Constant memories are read only memories that reside on the GDDR DRAM and are accessible to the host processor. Texture memory provides convenient indexing and filtering for certain applications thereby higher bandwidth can be achieved provided there is locality in the texture fetches.

Shared memory is the SRAM built into each of the Streaming Multiprocessors and serves as the high speed memory for data access. The shared memory is organized into multiple banks and avoiding bank conflicts is the key while accessing the shared memory. Shared memory is not accessible by the host and it is dedicated only to the cores in that SM. Shared memory is allocated either before the kernel call or at the time of the kernel call and it also doubles up as a L2 Cache.

2.7 Related Work

This section briefly describes some of the previous contributions in the area of the work presented here. This section is divided into three areas: one in the area of FPGA and heterogeneous computing based acceleration, second, acceleration of Smith-Waterman on GP-GPUs, third, the alternate methods to program the hardware accelerators.

2.7.1 Attempts for Hardware Acceleration of Bioinformatics Applications

Due to its data and compute intensive nature, several attempts have been made in recent years to port the bio-applications on custom hardware accelerators. While some of the work describe implementation on just the hardware platforms, the others are on heterogeneous systems like the Convey HC-1. These provide a good picture for porting bio-applications on to the application engines of Convey HC-1.

Altera's white paper on the implementation of Smith-Waterman [25] on XD1000 provides a good insight into the implementation of Smith-Waterman on a reconfigurable hardware. XD1000, a reconfigurable supercomputing platform similar to Convey HC-1, has a dual Opteron motherboard with Altera Stratix II FPGA inserted directly into one of the sockets. Due to its clear and concise description, this white paper is a good reference for hardware implementation. The work presented here derives several low level implementation ideas from this paper and it is described later in Section 3.1.

Another research paper presents the use of hybrid system for short read mapping employing both FPGA-based hardware and CPU-based software [26]. The CPU handles simple tasks of read coding, transferring the coded reads to the aligner along with its operating commands and receiving the alignment results. The alignment engine designed using the custom hardware on FPGA, accelerates the alignment process.

Several algorithms in sequence alignment that require BWT, use FM indexing. The work on the hardware implementation of string matching using FM-Index [27], is helpful in providing hardware acceleration to those applications that employ the use of BWT. It states that execution of Bowtie using this approach saw two orders of magnitude increase in speedup.

Another recent work involved the implementation of BFAST on Hardware, by University of Washington on the M501 board [28]. Of the five stages in the BFAST algorithm, two of the most time consuming stages, "match" and "local align", have been implemented in hardware. This led to two orders of magnitude speedup versus the BFAST software, consuming just 2.16% of the energy consumed by the BFAST software.

The extended Smith-Waterman using Associative Massive Parallelism, SWAMP+ [29], shows BLAST like sub alignments with Smith-Waterman like sensitivity. This was introduced in three different parallel architectures: Associative Computing (ASC), the ClearSpeed co-processor, and the Convey Computer FPGA co-processor. Here the unmodified Convey Smith-Waterman program, *cnysws*, was called in a python script to perform the sub alignments.

Grigorios Chrysos et al. [30] presented an in-dept look at the potential for FPGA in bioinformatics algorithms. Here the NCBI BLAST was profiled and the most time consuming part of the software was mapped on to the Convey HC-1's AEs and the results validated. Although it is yet to be optimized, it is expected to offer atleast one order of magnitude speedup against the NCBI implementation running on the Host Processor.

Bakos et al. [20] extended the BEAGLE API for Phylogenetic Likelihood Function (PLF) on Convey HC-1. These APIs include optimized algorithms for various parallel architectures. Due to a high arithematic intensity of 130 floating point operations per 64 bytes of data, this implementation gave 78 GFOPS of performance, which is a 40x speedup compared to the CPU implementation on the Convey host.

Oliver et al. [31] performed the SW algorithm on a standard Virtex II FPGA board, using a linear systolic array, achieving a peak performance of about 5 GCUPS using affine gap penalties.

Li et al. [32] used the Altera Nios II soft-processor to implement the algorithm. This was optimized by using the FPGA fabric to compute the value of each cell of the score matrix through a custom instruction. This accelerated the algorithm's computation time by up to 160 folds compared to a pure software implementation.

Allred et al. [33] demonstrated an implementation of the Smith-Waterman algorithm in a novel FSB module using the Intel Accelerator Abstraction Layer. They modified the SSEARCH35, an industry standard open-source implementation of the Smith-Waterman algorithm, to transparently introduce a hardware accelerated option to users. They demonstrated a performance of nine billion cell updates per second using this technique.

2.7.2 Work on GP-GPUs

Recently, many GP-GPU implementations of Smith Waterman [34] have been proposed with the advent of CUDA and general purpose computing on CUDA. Most of them involve finding the most similar sequence in a genomic database for a given query sequence. When necessary, the full alignment is obtained in CPU, which does not cause a bottleneck when the sizes of the sequences are small.

One of the early CUDA Smith-Waterman implementations [35], saw the potential of GP-GPU in genome alignment and it was claimed to be 2 to 30 times faster than the implementations of Smith-Waterman on commodity hardware. Speeds of more than 3.5 GCUPS (Giga Cell Updates Per Second) was achieved on a workstation running two GeForce 8800 GTX.

Smith-Waterman programs like CUDA SW++ [36] optimizes the algorithm using various GPU centric techniques such as coalesced access of the global memory and the use of texture memories for a faster access. The same was further improved using SIMT abstraction and partitioned vectorized algorithm in CUDA SW++ 2.0 [15] to achieve up to 1.76 times performance improvement compared to CUDA SW++. Finally their latest version, CUDA SW++3.0 [34] uses GPU PXT SIMD instructions and CPU SSE instructions to achieve a speedup of around 2.9 over CUDA SW++ 2.0 on NVIDIA Kepler Architecture.

However some of the recent work involved dealing with specific problems like aligning longer sequences and dealing with trace-back on GP-GPUs.

A GPU algorithm, CUDAlign [37], that is able to compare Megabase biological sequences with an exact Smith-Waterman affine gap variant was proposed. CUDAlign was implemented in CUDA and tested on two GPU boards, separately. A peak performance of 20.375 GCUPS was reported showing potential scalability of this approach.

Later in 2011, the same authors obtained alignment on GPU in linear space [38]. Here, the emphasis was to obtain a complete alignment for long sequences on GPU. This work solves a specific case of aligning a long genome sequence to another in steps to conserve the memory usage.

Another recent work [39] introduces stripped and chunked alignments to align a pair of long sequences. They reduced the shared memory usage, global memory per SM and the I/O traffic between them, in turn achieving an order of magnitude reduction in run time relative to competing GPU algorithms.

Sequence Alignment on the PlayStation-3, CBESW [40], was designed for the Cell/BE-based PlayStation-3 (PS3) by Warawan et al. This achieved a peak performance of about 3.6 GCUPS demonstrating that the PlayStation-3 console can be used as an efficient low cost computational platform for high performance sequence alignment applications.

Another implementation of Smith-Waterman on Cell-Broadband Engine [41] involved scaling of the algorithm across multiple Cell-BE nodes on both the IBM QS20 and the PlayStation-3 Cell cluster platforms to achieve a maximum speedup of 44, when compared to the execution times on a single Cell node. Here, the Smith-Waterman algorithm was also optimized on GP-GPU by including optimal data layout strategies, coalesced memory accesses and blocked data decomposition techniques to achieve a maximum speedup of 3.6 on the NVIDIA GP-GPU when compared to the performance of the naive implementation of Smith-Waterman.

2.7.3 Unconventional Hardware Programming

The project ConVI, described in Chapter 4, was inspired by an application called Bflow [19], developed a year ago. Bflow uses a specific graphical entry tool tailored for hardware design, called Azido. The hardware designed from Azido was deployed on Convey HC-1 and was interfaced with the Azido's front-end graphical interface. Below is the the Azido-Convey interface involving the use of COM objects to communicate with the telnet server in Bflow. The MPIP is the telnet server that communicates with the management processor.

```
Azido <=> COM Object <=(SSH Tunnel)=> Relay (Convey) <=> MPIP Telnet Server
<=> Management Interface (CSR Ring)
```

The COM object makes a connection to a local socket, which is tunneled to a port on the Convey box, on which the azprobe utility listens.

Another approach to programming accelerators involved the use of OpenCL [42]. OpenCL libraries were used to program the Convey HC-1 system. This work was later extended to incorporate systolic arrays and Smith-Waterman in hardware [43]. This approach has the high level programming characteristics against which the current proposed work can be compared.

Chapter 3

High Performance Smith-Waterman Implementation

In this implementation, an attempt has been made to harness the re-configurable hardware of the Convey systems. Dynamic programming was chosen as it maps efficiently with the hardware and for all the advantages discussed in Section 1.1. The design was implemented using the bottom-up approach.

At first, a single cell of Smith-Waterman Processing Element (PE), is built in hardware. The design is then extended to a cascade of PEs to form a systolic array. Then the systolic arrays are duplicated to occupy the entire fabric of the FPGAs. Every systolic array is connected to the memory controllers through the Convey-provided FIFOs. The AEs are finally interfaced to the dispatch interface to receive the source and destination pointers. The flow of data to and from the dispatch interface, memory controllers and the systolic arrays are controlled by a state machine.

3.1 Smith-Waterman Processing Element

The Processing Element (PE) is the smallest module in this design. Each PE controls the value of a single cell of the scoring matrix at a time. The block diagram of the PE is shown in Figure 3.1. It takes values from three neighboring cells in the scoring matrix, computes their respective scores, compares the results and stores the maximum.

Each PE then moves on to compute the value of the cell that is immediately below the cell that was just computed. Different PEs have the task of computing the values of the cells in different columns in the scoring matrix. The PEs are cascaded to one another as shown in Figure 3.2. Each PE gets the value of the neighboring cells to the cell being executed as follows,

- Value of upper cell from itself, delayed by a clock cycle.
- Value of the left cell from the adjacent PE.
- Value of the diagonal cell from the adjacent PE delayed by a clock cycle.

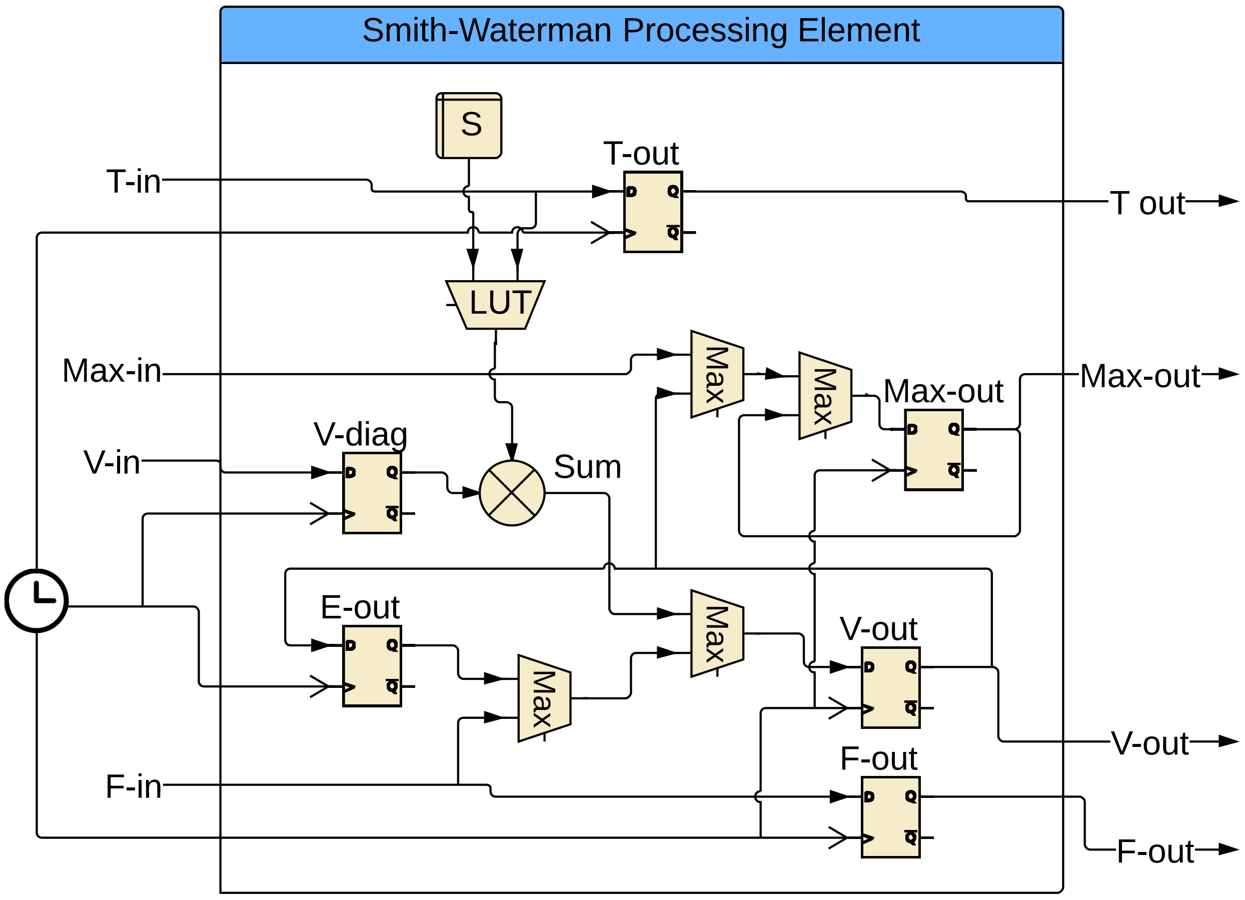


Figure 3.1: A single Smith-Waterman Processing Element.

3.2 Systolic Array

The cascade of PEs continue such that the number of PEs in the cascade is equal to the length of the query sequence. This cascade of PEs computes the entire Smith-Waterman scoring matrix in hardware and its termed as the ”systolic array”. The systolic array is a module thats holding the Smith-Waterman PEs together. Each application engine has several independent systolic arrays, each capable of computing 32 base pair sequence alignments.

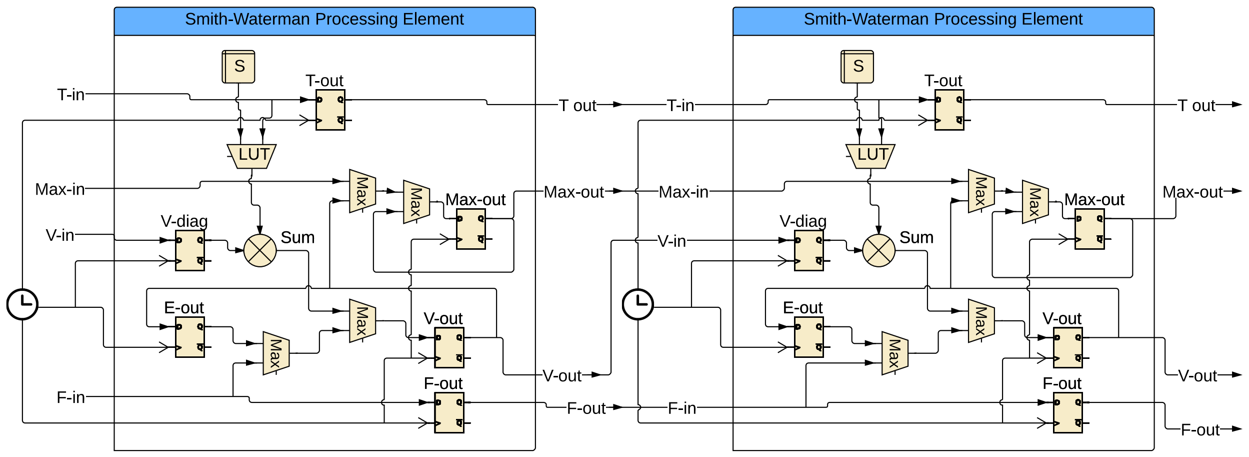


Figure 3.2: Cascading of the Smith-Waterman PEs.

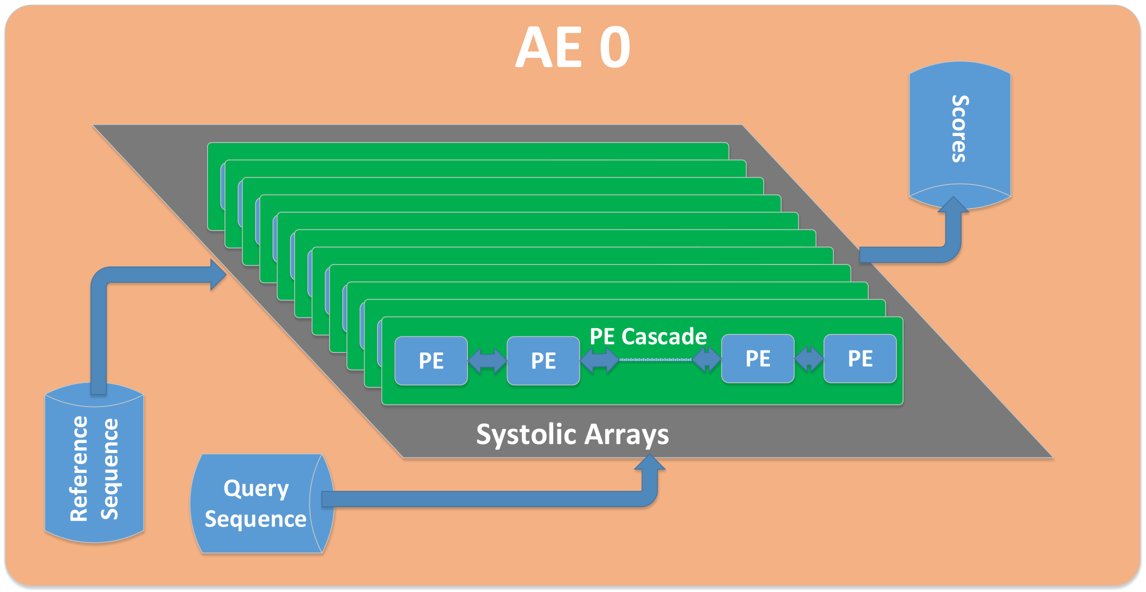


Figure 3.3: Hierarchical design of PEs and their interface to FIFOs in each of the AEs.

Each column in the score matrix is controlled by the same PE. This implementation is based on the white paper by Altera Corp [25].

Each application engine in Convey has multiple systolic arrays to compute the scores of multiple sequences in parallel. This was done to optimize the resource usage in the FPGA while utilizing the maximum memory bandwidth. Figure 3.3 shows each application engine with systolic arrays where each systolic array has its own independent memory interface.

3.3 FIFO Interface

Convey co-processor memory supports DDR2 DRAM modules. These memory modules are interfaced to the co-processors through eight memory controllers as explained in Section 2.4.5. The data between the memory controllers and the AEs can be queued in Convey supplied FIFO modules to streamline the memory accesses. The FIFOs use either the Block-RAMs available on the FPGAs or SRAM in the LUTs. This can be configured based on the depth of the FIFO required and the speed required by the application. Three FIFOs are used in this design for each one of the systolic arrays. One FIFO is used to queue the reference sequence, another is used to queue the query sequence and the last FIFO is used to queue the scores before they go back into the memory. Figure 3.4 shows the FIFO interface provided in the Convey PDK.

```
fifo #(.DEPTH(32), .WIDTH(64), .AFULLCNT(28)) op1_fifo (
 .clk    (clk),          // clock to fifo
 .reset  (r_reset),      // reset to fifo
 .push   (op1_wea),           // push operand 1
 .din    (r_mc_rsp_data),     // data in
 .afull  (op1_fifo_afull),    // almost full
 .full   (),
 .cnt    (),
 .oclk   (clk),       // output clock
 .pop    (pop_fifo),          // pop signal
 .dout   (op1_ram_out),       // output to ram
 .empty  (op1_qempty),        // empty fifo
 .rcnt   ()
);
```

Figure 3.4: FIFO Interface

These FIFOs interface to the eight independent memory controllers depending on the crossbar configuration. Turning on the crossbar makes any memory controller accessible to the FIFOs in any of the AEs. This might improve the bandwidth but requires proper packing of sequences in the co-processor memory. Figure 3.5 shows the interconnections between the FIFOs in each of the AEs to their respective memory controllers when crossbar is disabled.

3.4 Dispatch Interface

In the Smith-Waterman application, the dispatch consists of three pointers to the co-processor. Two of the pointers point to the memory location where the query and the reference sequences

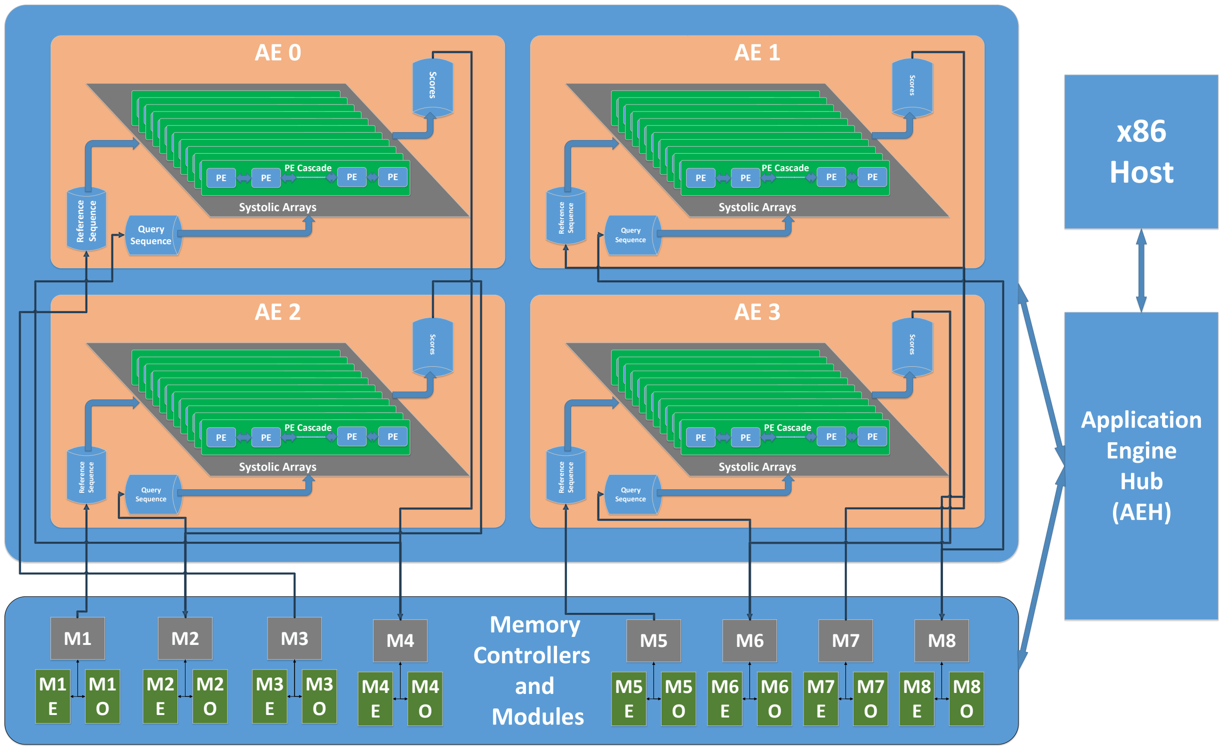


Figure 3.5: Final structure of the Smith-Waterman design showing the connections to the memory and the dispatch on Convey HC-1.

are stored in the co-processor memory and the third pointer points to the location in the co-processor memory where the resultant scores are stored. A blocking copcall is made to pass these parameters to the AEs through the dispatch interface. The host processor stalls until the AEs complete the execution and returns control to the host. The host application then copies the scores from co-processor memory and writes it to a file as shown in Figure 5.4.

3.5 Interface to the Bio-applications

In this work, one of the primary goals of the Smith-Waterman application is to provide accelerated dynamic programming API to any bio-application that has a dynamic programming phase. Many bio-applications can be partitioned into segments where the dynamic programming portion of the application is ported into Convey co-processor through the copcalls, while the rest of the code can reside and execute on the host processor.

3.6 General Purpose-GPUs

The design for Smith-Waterman is implemented in NVIDIA GP-GPUs, GTX 580 and GT 440 using the Fermi architecture. This work began with two implementations for computing the score matrix. One where each thread computes the score for a single cell and the other where each thread computes the score for the an entire row of the Smith-Waterman score matrix. In both cases the implementation was first done on a global memory and then transferred to a shared memory location for a faster memory access. CUDA 5.0 was used for these implementation.

Since only the maximum scores are tracked, the transfer of the score matrix back and forth between host and the GPU is not required and the matrix is used as a scratch pad.

3.6.1 One Thread per Cell

This is a simpler implementation as shown in Figure 3.6 where each thread continuously updates a single cell of the matrix. As each cell is processed by its own thread, the indexing is simpler and there is no change in the index with iterations over a kernel call.

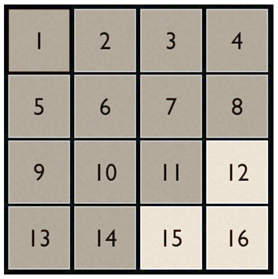


Figure 3.6: One Thread per cell.

However this implementation has an inherit disadvantage that most of the threads are idle. For a $m - length$ query and a $n - length$ reference, the total number of threads required is $m * n$ while the number of threads in use is limited to the size of the query m. Thus, at any given time, $(m - 1) * n$ threads are idle. The thread requirement also scales quadratically with the query and the reference sizes.

3.6.2 One Thread per Row

This is one of the solutions to utilize threads effectively. Here the entire row of cells is computed by a single thread. Hence the number of threads launched is equal to the number of rows which is in turn equal to the query length m. In Figure 3.7, the thread number computing each cell is shown, the thread number is constant across a row as seen.

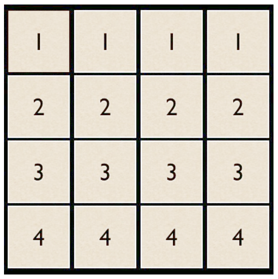


Figure 3.7: One Thread per Row.

This approach is similar to the Altera's implementation [25] on a FPGA. This approach has an index update with each iteration within a kernel. The number of threads are fixed and are equal to the size of the queries. The reference sequences of any size can be used as the number of threads does not depend on the size of the reference sequences.

Chapter 4

Convey-LabVIEW Interface

This work plays a part in the development of Convey with Virtual Instruments (ConVI) design flow. Here, the LabVIEW programming environment is demonstrated as a viable platform to build bio-applications on Convey. This work tries to simplify the task of using heterogeneous systems with re-configurable devices for bioinformatics researchers.

While LabVIEW offers one of the potential design environment for non-hardware design engineers, it is naively supported to only run on National Instruments FPGA targets, called the Re-configurable I/O (RIO) devices. It is built to work on embedded platforms and not on data centers. Thus, comes the challenge to interface LabVIEW to high performance system such as Convey. The Interactive Front Panel communication on LabVIEW uses a polling-based method of communicating between the Host VI and the FPGA VI by reading and writing indicators and controls [22]. This feature also calls for a robust low latency communication channel between the host VI and the FPGA VI.

To accomplish this task, this work builds on the existing work on the Bflow application working on the Azido-Convey interface. Going further, the JTAG interface is exploited to enable the remote access of the application from the LabVIEW front panel.

4.1 LabVIEW FPGA Communication

The LabVIEW front panel communicates with FPGA with constant monitoring of the status, command and data registers in its generated HDL. An Host VI can control and monitor data passed through the FPGA VI front panel. The values that do not have controls or indicators cannot be accessed on the wires on the FPGA VI block diagram unless the data is stored in a DMA FIFO. Read/Write Control function reads and writes controls and indicators in the order they appear in the Read/Write Control function on the block diagram [22]. The FPGA Module creates a register map, specific to the FPGA VI, that includes a hardware

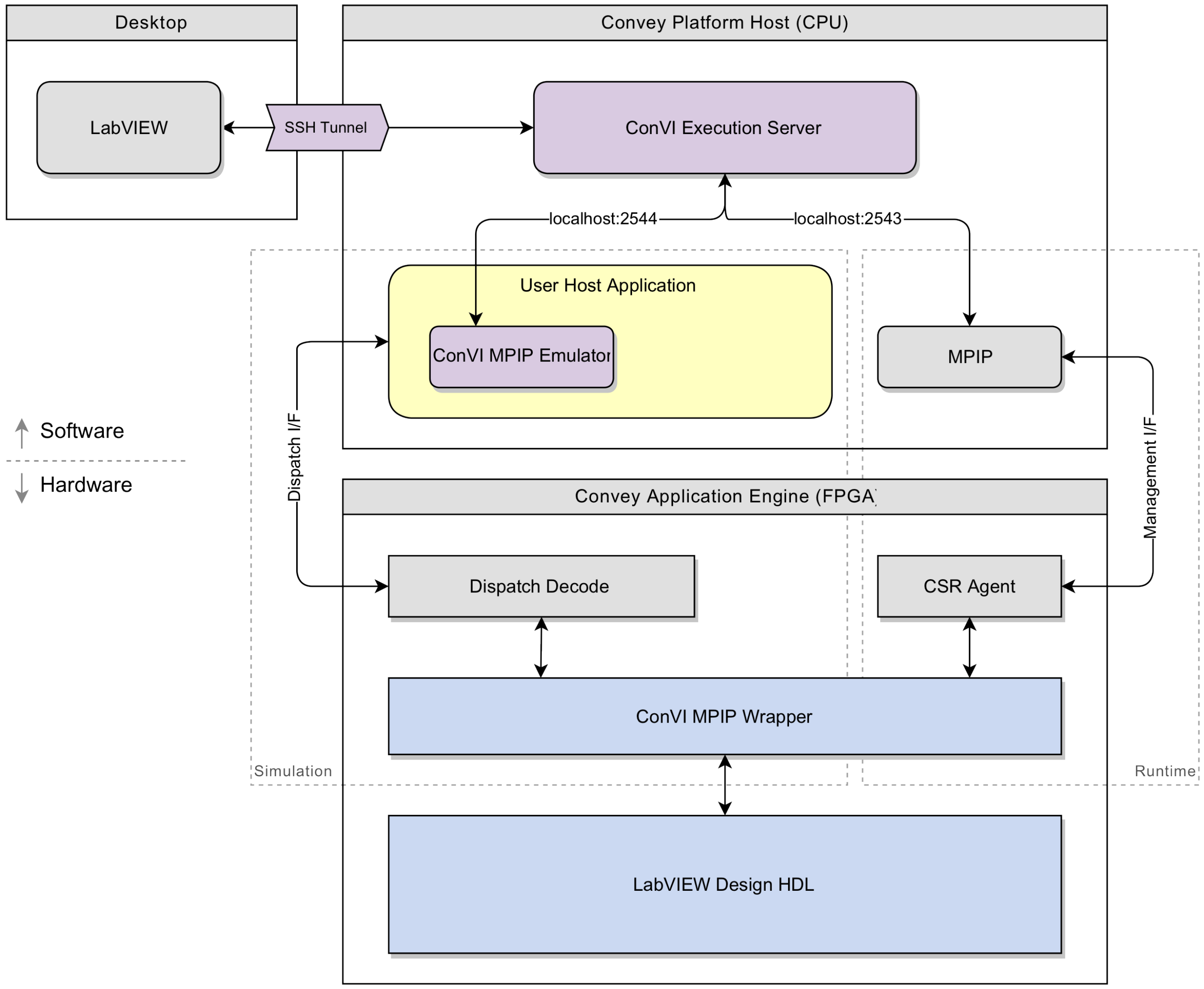


Figure 4.1: ConVI Overview.

register for every control and indicator.

4.2 Connections and Bit-Mapping

Here, the Convey's management interface is used to connect the signals to the generated LabVIEW HDL though a wrapper logic. The CSRs are written through the telnet commands at the host from the LabVIEW server. The data in CSRs are passed to the LabVIEW HDL through its data input ports. The output from the LabVIEW HDL is written to a CSR which is retrieved by a CSR read command. This setup is used for run-time communication

Table 4.1: Request and Response formats

Request		
Bits	: 00 − 32	: 4-byte data for the LabVIEW registers
Bits	: 33 − 42	: 11-bit Address of the LabVIEW registers
Bit	: 48	: Read Strobe
Bit	: 52	: Write Strobe
Bits	: 56 − 57	: 2-bit Message ID

Response		
Bits	: 00 − 32	: 4-byte data for the LabVIEW registers
Bit	: 48	: Ready
Bit	: 52	: Data Valid
Bits	: 56 − 57	: 2-bit Message ID

between Convey and the LabVIEW front panel.

The bit encoding for both CSR and AEG registers, that carry the commands from the MPIP telnet server to the LabVIEW HDL are shown in Table 4.1 and in Table 4.2.

Here, the Message ID can take four values: 0,1,2,3 and they need to be cycled between consecutive messages to be taken in as a new request. The Message ID for the response will be the same as the message ID for the request.

4.3 Simulation of ConVI

During simulation, the AEG registers in the dispatch are used in place of CSR for the transactions due to the lack of interface from the simulator to the management interface. The AEG registers are written through non-blocking copcalls thereby not affecting the host application code running on the x86 processor. Figure 4.2 shows the Convey assembly code implementing a non-blocking copcall to write and read values from AEG registers that connect to the ports of LabVIEW HDL.

4.4 LabVIEW HDL Wrapper

A wrapper is a piece of code that abstracts the use of another piece of code. Wrappers are used to adapt an existing design to have a different interface. Due to the heterogeneous

```
##-----------------------------------
    .globl  lvload
    .type   lvload. @function
    .signature pdk=4

lvload:  # function that loads the personality and polls for read/write requests
  loop:
   ld.uq $run(%a0),%a20      # Exit the loop if run = 0
   cmp.uq %a20, %a0, %ac0 # Check for loop complete
   br %ac0.eq, done  # Stop

        ld.uq $mode(%a0),%a22 # load the mode register to decide which action to
    be performed

   cmp.uq %a22, $0, %ac0   # 0 for polling back for a different mode
   br %ac0.eq, loop  # Poll

   br loop                     # stall
##-----------------------------------
 write_lv:
   ld.uq $Payload_In(%a0),%a30   # load the payload into a30
   mov.ae0 %a30,$0,%aeg            # move the payload into aeg0 to dataport in
   st.uq %a0, $mode(%a0)           # change mode back to 0 for polling
   br loop                         # stall and wait for next request

  read_lv:
   mov.ae0 %aeg,$0,%a30            # read the dataport out through aeg0
   st.uq %a30,$Data_Out(%a0)       # store the value into Data out
   st.uq %a0, $mode(%a0)           # change mode back to 0 for polling
   br loop                         # stall and wait for next request
##-----------------------------------
  done:
        rtn

lvloadEnd:
        .globl  lvload
        .type   lvload. @function
        .cend
##-----------------------------------
```

Figure 4.2: Non-blocking copcall in Convey assembly code

Table 4.2: The Bit Mapping for LabVIEW Commands through 64-bit CSR and AEG Registers.

64 bits of the CSR/AEG registers								CSR 0x8007	CSR 0x8008	Mask
7	6	5	4	3	2	1	0	Data-LSB	Data-LSB	FF
15	14	13	12	11	10	9	8	Data-LSB	Data-LSB	FF
23	22	21	20	19	18	17	16	Data-MSB	Data-MSB	FF
31	30	29	28	27	26	25	24	Data-MSB	Data-MSB	FF
39	38	37	36	35	34	33	32	Address-LSB	X/X	FF
47	46	45	44	43	42	41	40	X/Address-MSB	X/X	07
55	54	53	52	51	50	49	48	Write/Read	Valid/Ready	11
63	62	61	60	59	58	57	56	X/Msg-ID	X/ Msg-ID	03

nature of Convey, LabVIEW does not have a built-in support to interface the peripherals of the Convey computers. Hence, a wrapper is used to interface the LabVIEW generated HDL to Convey's top-level HDL called *cae_pers*. Figure 4.3 gives the block diagram of the wrapper. The wrapper serves the purpose of interfacing the HDL generated from LabVIEW to the Convey environment. The LV HDL has three input ports, and three output ports. Each of the three pairs of input/output ports are used to interface to the memory, dispatch and the management interfaces. As shown in Figure 4.3, the wrapper consists of these parts:

1. A stub to connect to all the I/O ports coming out of LabVIEW HDL that connects to the originally intended Spartan-3E board: This stub simulates the environment of the Spartan-3E board for the LabVIEW HDL.

2. An interface to the streamer source and sink HDLs to stream data in and out of the co-processor memory: The input port interfaces with the source streamer that streams the data from the co-processor memory into FPGAs, which in turn is streamed to the LV HDL block. This interface is used to fetch the sequences from memory. The output port interfaces with the sink streamer which streams the alignment scores generated in the LV HDL to the co-processor memory.

3. Interface to the AEG registers through the I/O in the modified XML resource file: These are used to interface the LV HDL to the dispatch interface. These are used to transfer data from the host application to the LV HDL. In this application, the input data can be a pointer to the memory location where the sequences are stored.

4. Interface to the BPORT IN and BPORT OUT ports, which are the interfaces to the LabVIEW HDL from the VI Front panel: These are interfaced to the management processor. Here the input port is used to send the commands to read/write the registers

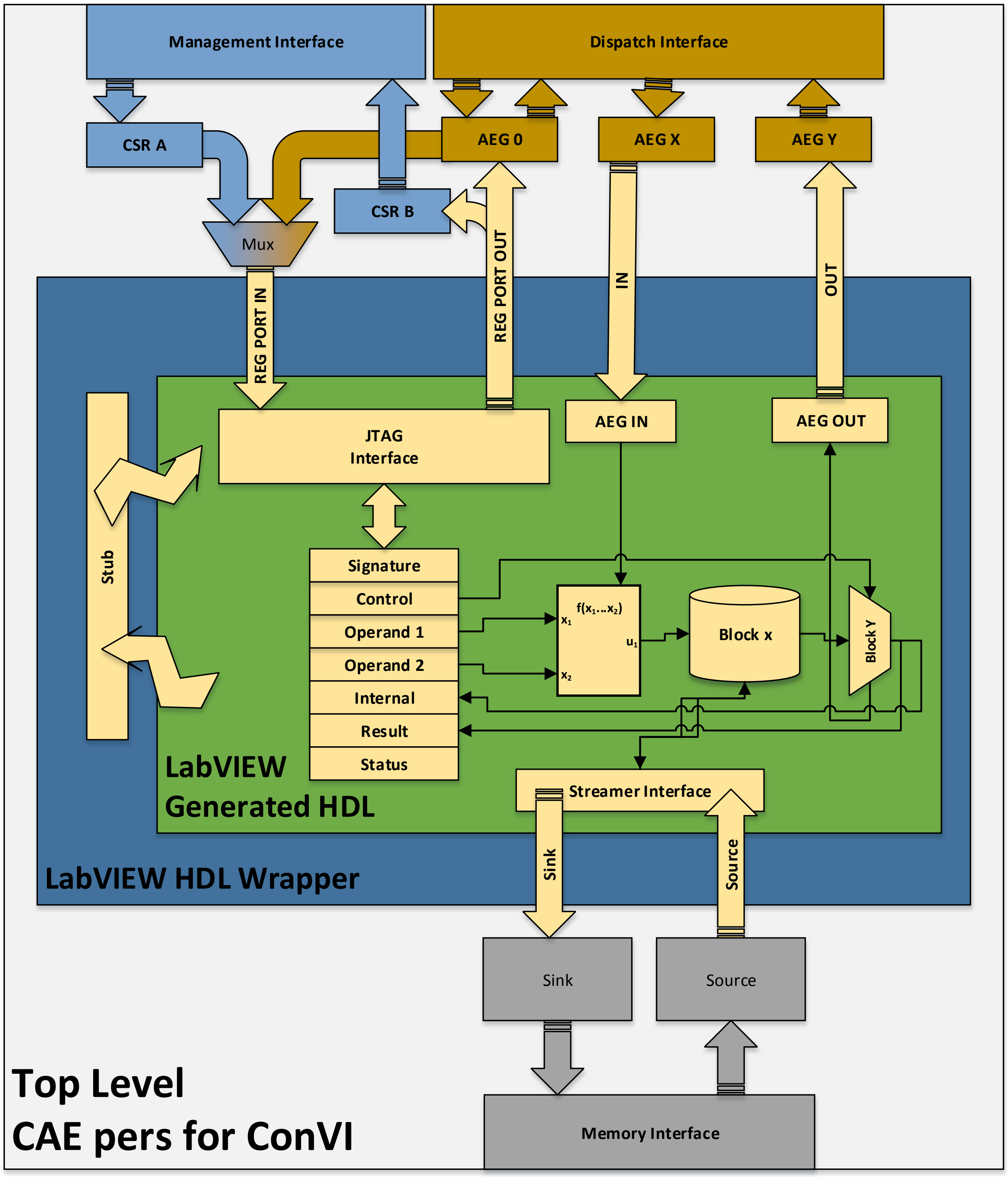


Figure 4.3: Wrapper interfacing LabVIEW-generated HDL to Convey's top-level CAE pers.

from the LabVIEW front panel and an output port is used to send the data back to the front panel as explained in Section 4.2.

4.5 Designing using LabVIEW

After designing the LabVIEW-Convey communication channel and the wrapper, LabVIEW is used for designing applications for the hardware accelerator. To demonstrate this design flow using LabVIEW, a 16-PE Smith-Waterman pipeline is designed in LabVIEW. First, a single PE of Smith-Waterman is designed using the LabVIEW graphical programming language as shown in Figure 4.4. This PE is replicated using LabVIEW VI scripting [44] to generate a 16-PE pipeline as shown in Figure 4.5. Running this application generates the HDLs for this design, which are copied to a directory with other files from Convey PDK for simulation and synthesis using Convey tools. This design runs on the Convey HC-1 accelerator and communicates with the LabVIEW front panel thereby providing the user to control and monitor the accelerator using the LAbVIEW's graphical interface as shown in Figure 4.6. Thus, the complex job of designing the accelerator for Convey HC-1 is simplified by using LabVIEW.

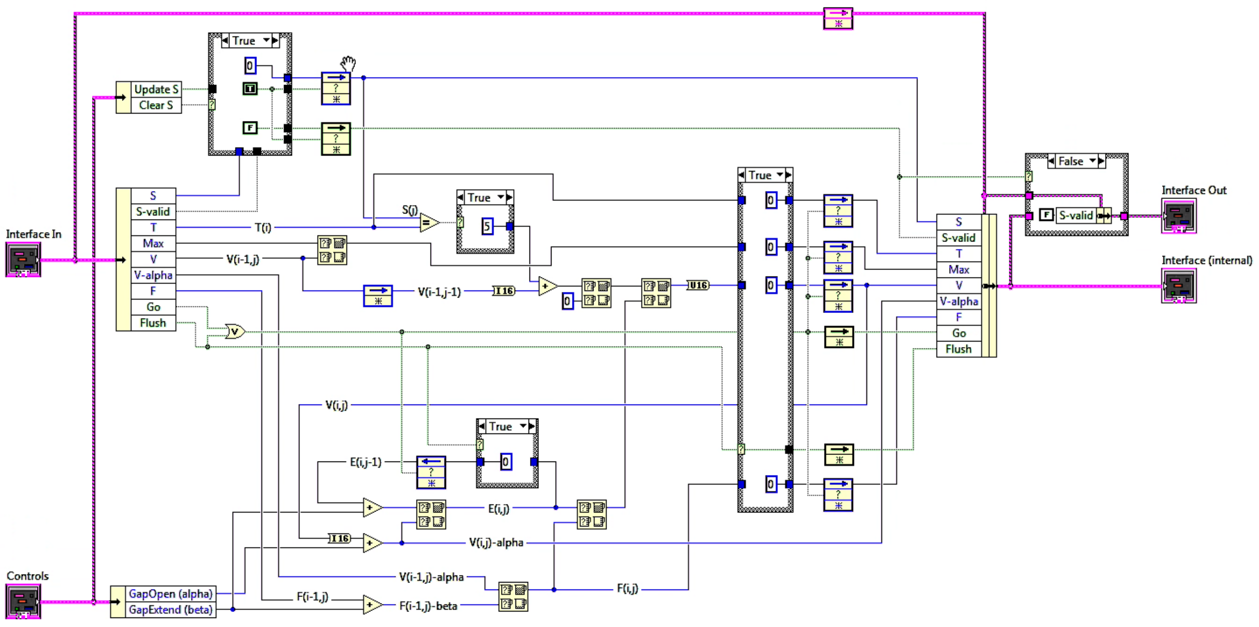


Figure 4.4: Smith-Waterman PE designed using LabVIEW.

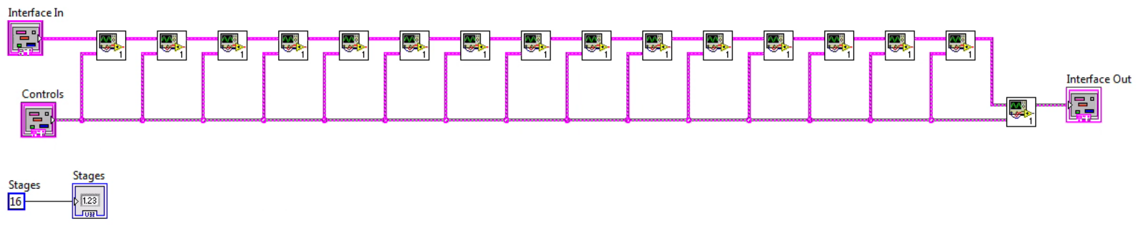


Figure 4.5: A 16-PE pipeline generated using LabVIEW VI Scripting.

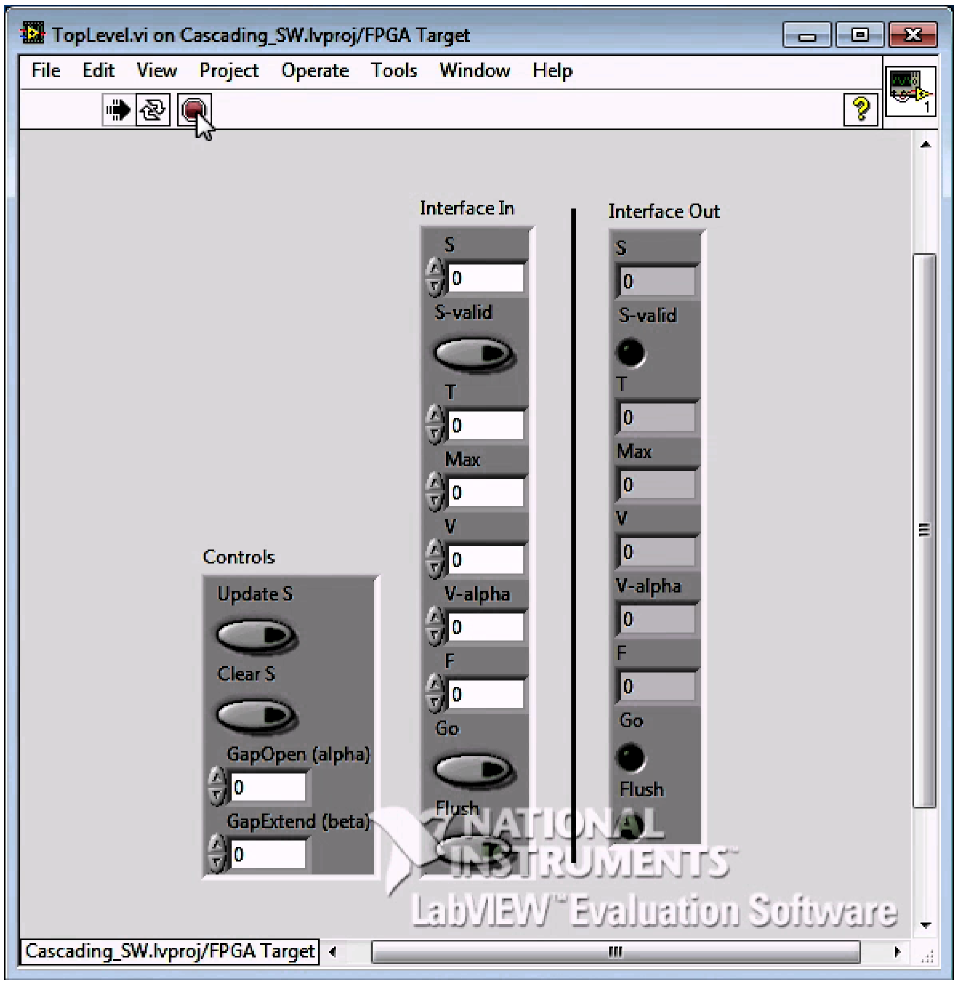


Figure 4.6: LabVIEW front panel with controls and indicators for Smith-Waterman application running on Convey HC-1.

Chapter 5

Results and Analysis

This chapter focuses on the results of the high performance Smith-Waterman implementations and the implementation of LabVIEW's interface to Convey HC-1. This chapter is divided into the following sections. The first and the second sections deal with the results of the high performance Smith-Waterman on Convey HC-1 and NVIDIA GP-GPU respectively. Then the performance of these implementations are compared with the recent work done in this area. Finally, the results of the LabVIEW based interface to Convey, called ConVI, is discussed.

5.1 High Performance Smith-Waterman

The Smith-Waterman algorithm was designed for the Convey HC-1 accelerator with a single Smith-Waterman PE first, then the design was cascaded into a pipeline to form a complete systolic array. The pipelines were replicated as much as possible to take the complete advantage of the available logic resources and the memory bandwidth in the system.

As the number of PEs in a FPGA directly translates to the performance, emphasis here is to maximize the number of PEs contained in each of the AEs. Hence, the Section 5.1.1 deals with the area and resources in the fabric used for building the accelerator. The amount of data consumed per cycle by the application engines is directly proportional to the number of PEs in the design; thus, a theoretical estimate of the bandwidth is presented in the Section 5.1.2. Finally, this design is tested by simulating it using the Convey's HDL simulator. The waveforms in the timing diagram are analyzed using the Synopsys DVE tool to verify the assumptions. These are reported in the Section 5.1.3.

5.1.1 Area Occupancy

In the implementation of the accelerator, the number of PEs are increased in steps to accommodate the highest number of PEs in each of application engine. Several designs were attempted before arriving at an optimal configuration for the Convey HC-1.

Although the co-processor board has several non-user programmable FPGAs for memory controllers, dispatch and other modules, a small portion of the application engine fabric is used to create the interfaces to these modules. Another portion of the fabric is used in the implementation of the skeleton that supports the PEs and seamlessly connects them to the memory and the dispatch interfaces. The remainder of the the fabric area is used for building the PEs in the systolic array.

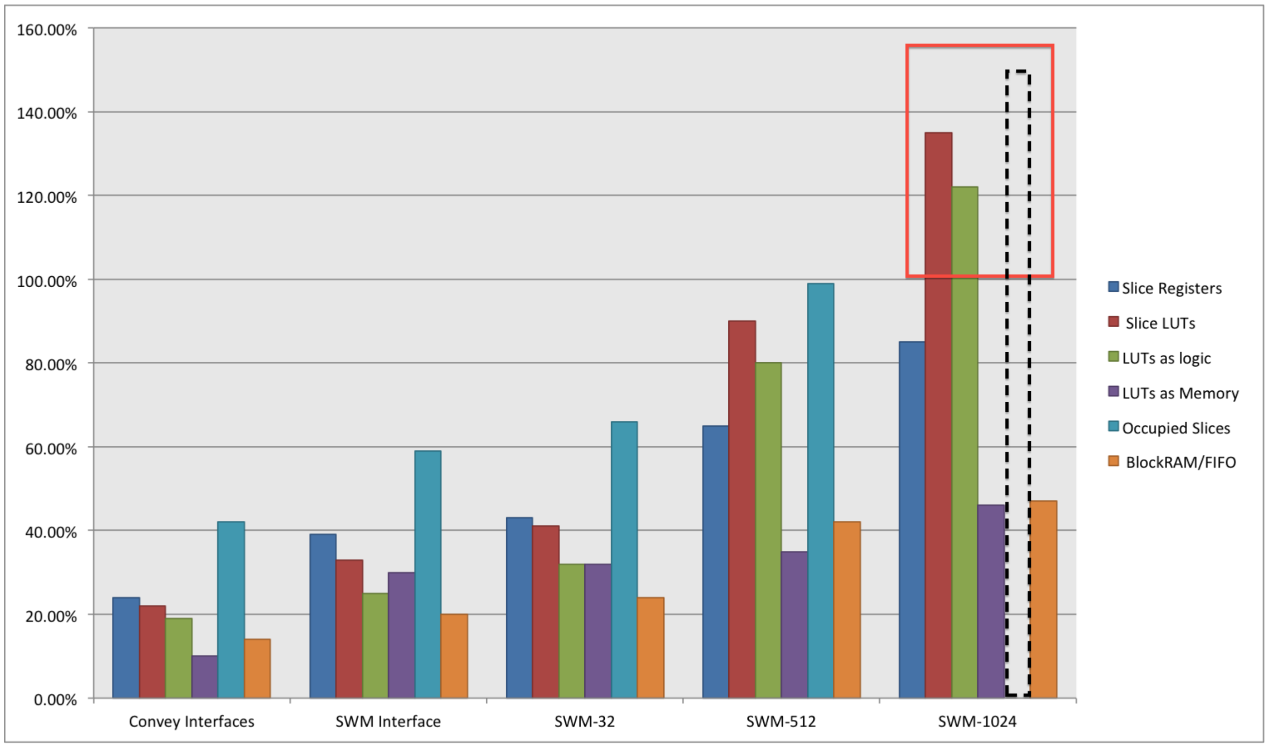


Figure 5.1: Resource consumption by the Smith-Waterman accelerator on Convey HC-1.

The resource consumed on the fabric for different implementations are shown in Figure 5.1. The resource consumption on the FPGA is evaluated in terms of Slice Registers, Slice LUTs, the percentage of LUTs that are used for logic in the design, the percentage of the LUTs that are used as memory, the total number of slices used and finally the amount of BRAM used in the design. As seen in the figure, the pre-existing interfaces that connect to the other modules in co-processor board occupy 20% – 40% of the total fabric available on the XC5VLX330 FPGA. A skeleton to support 32 PEs per FPGA, increased the occupancy to

40% − 60%. This design was populated with 32 PEs to increase the occupancy by 5% − 10% greater than the existing skeleton. This structure was replicated in the design until it could support 512 PEs per AE with an occupancy of 80% − 99%.

This implementation involved a pipeline with 32 processing elements, and 16 pipelines per AE, giving rise to $16 * 4 * 32 = 2048$ processing elements to achieve a peak theoretical throughput of 2048 PE * 150 MHz = 307 GCUPS (Giga Cell Updates per Second). The synthesis report of this design on the AEs is attached below. The report shows that this is a proper fit for the XC5VLX330 FPGA.

```
Release 14.1 Map P.15xf (lin64)
Xilinx Mapping Report File for Design 'cae_fpga'

Design Information
------------------
Command Line   : map -intstyle xflow -pr b -detail -ignore_keep_hierarchy -xe n
    -w -t 1 -ol high -o cae_fpga.ncd
cae_fpga.ngd
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -2
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Wed Apr 17 19:28:48 2013

Design Summary
--------------
Number of errors:      0
Number of warnings:  425
Slice Logic Utilization:
  Number of Slice Registers:             136,336 out of 207,360   65%
    Number used as Flip Flops:           135,792
    Number used as Latches:                  544
  Number of Slice LUTs:                  187,057 out of 207,360   90%
    Number used as logic:                166,244 out of 207,360   80%
      Number using O6 output only:       127,283
      Number using O5 output only:         5,744
      Number using O5 and O6:             33,217
    Number used as Memory:                19,578 out of  54,720   35%
      Number used as Dual Port RAM:       19,290
        Number using O6 output only:       2,098
        Number using O5 output only:       1,153
        Number using O5 and O6:           16,039
      Number used as Shift Register:         288
        Number using O5 and O6:              288
    Number used as exclusive route-thru:   1,235
  Number of route-thrus:                       7,307
    Number using O6 output only:               6,866
    Number using O5 output only:                 328
    Number using O5 and O6:                      113

Slice Logic Distribution:
  Number of occupied Slices:                  51,652 out of  51,840   99%
  Number of LUT Flip Flop pairs used:        200,442
    Number with an unused Flip Flop:          64,106 out of 200,442   31%
    Number with an unused LUT:                13,385 out of 200,442    6%
    Number of fully used LUT-FF pairs:       122,951 out of 200,442   61%
    Number of unique control sets:             4,732
    Number of slice register sites lost
      to control set restrictions:             8,591 out of 207,360    4%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice. A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                         861 out of   1,200   71%
    Number of LOCed IOBs:                        861 out of     861  100%
    IOB Flip Flops:                            1,016
    IOB Master Pads:                              20
    IOB Slave Pads:                               20

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                       122 out of     288   42%
    Number using BlockRAM only:                  122
    Total primitives used:
      Number of 36k BlockRAM used:                97
      Number of 18k BlockRAM used:                33
    Total Memory used (KB):                    4,086 out of  10,368   39%
  Number of BUFG/BUFGCTRLs:                       26 out of      32   81%
    Number used as BUFGs:                          2
    Number used as BUFGCTRLs:                     24
  Number of IDELAYCTRLs:                          18 out of      34   52%
  Number of DCM_ADVs:                              1 out of      12    8%
    Number of LOCed DCM_ADVs:                      1 out of       1  100%
  Number of ISERDESs:                            356
  Number of OSERDESs:                             20
  Number of PLL_ADVs:                         2 out of       6   33%
    Number of LOCed PLL_ADVs:                 2 out of       2  100%
  Number of SYSMONs:                          1 out of       1  100%

  Number of RPM macros:           72
Average Fanout of Non-Clock Nets:             3.65

Peak Memory Usage:  5857 MB
Total REAL time to MAP completion:  3 hrs 25 mins 30 secs
Total CPU time to MAP completion:   3 hrs 24 mins 33 secs
```

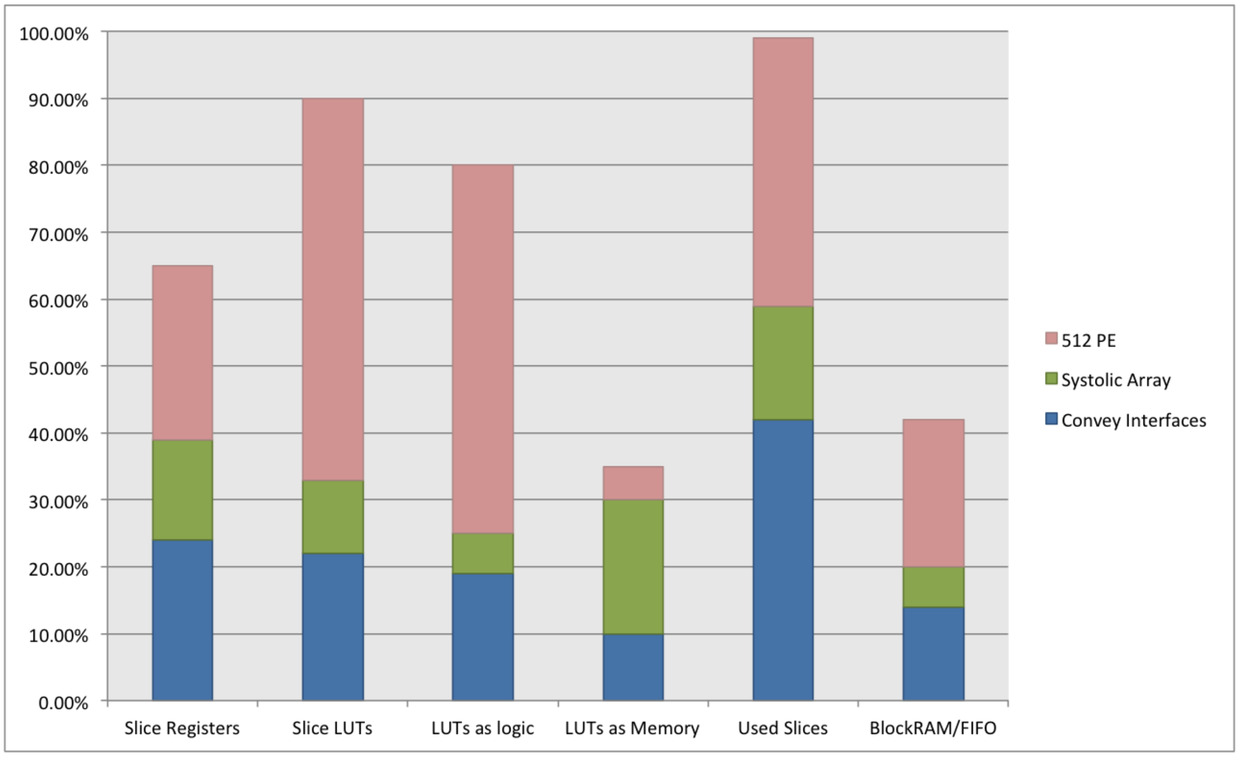


Figure 5.2: Resource allocation for each of the components in the design.

Figure 5.2 shows the allocation of resources for each of those components in the optimum design for HC-1 with 512 PEs per AE. It can be seen that the major chunk of the fabric is occupied by the 512 PEs. Most of the LUTs (55%) are used to implement logic functions and few of them (5%) are used for memory in the PE designs while the opposite is true in the case of the systolic array skeleton. The Convey-provided interface has up to 40% of used slices as seen before.

Figure 5.1 also shows a final implementation with 1024 PEs per FPGA. This attempt involved a pipeline with 64 processing elements, and 16 pipelines per AE, giving rise to $16{*}4{*}64 = 4096$ processing elements to achieve a peak theoretical throughput of 4096 PE * 150 MHz = 614

GCUPS (Giga Cell Updates per Second). The red box in the figure indicates that the design goes out of bounds for the XC5VLX330 FPGA. However this design can be used to program the Convey HC-1ex platform, since it contains the XC6VLX760 FPGA, which has more than twice the number of resources as the XC5VLX330 [45].

5.1.2 Bandwidth Requirements

As both HC-1 and the HC-1ex have the same memory bandwidth capacity, both the platforms have the same calculation.

Inputs

Each PE requires 4 bits from memory: 2 for query and 2 for database.
Each PE requires a single byte character from the block RAM for scoring.
Each PE requires a 2-byte short integer max score and five scores for computation.

Outputs

Each PE sends out a 3-bit direction vector to memory.
Each PE sends out its own value.

Assumptions

Each PE update is done in one clock cycle.
The design runs at maximum clock speed of 150 MHz.
The entire query sequence resides on the FPGA after a initial phase.
With striding enabled, the peak bandwidth available = 20 GB/sec [1].
Aggregate bandwidth for 4 AEs = 640 Gigabits/sec.

AE-Memory bandwidth required without trace-back

Number of bits per cycle per AE = 4 bits.
Number of bits for all the 4 AEs = 16 bits.
Therefore bandwidth required = $150 * 10^6 * 16 = 2.4$ Gigabits/sec.
This fits well within the maximum supported bandwidth.

Bandwidth required to store the trace-back vectors

Number of bits to send the directions information per cell = 3 bits.
Bandwidth required for Trace-back Vectors = $150 * 10^6 * 3 * 2048 = 921.6$ Gigabits/sec.
This is a high value to be practically incorporated on Convey HC systems.
Hence, trace-back is not used.

Thus, this is a compute-bound problem in Convey when trace-back is not performed in hardware. With trace-back, memory bandwidth becomes the bottleneck in the system.

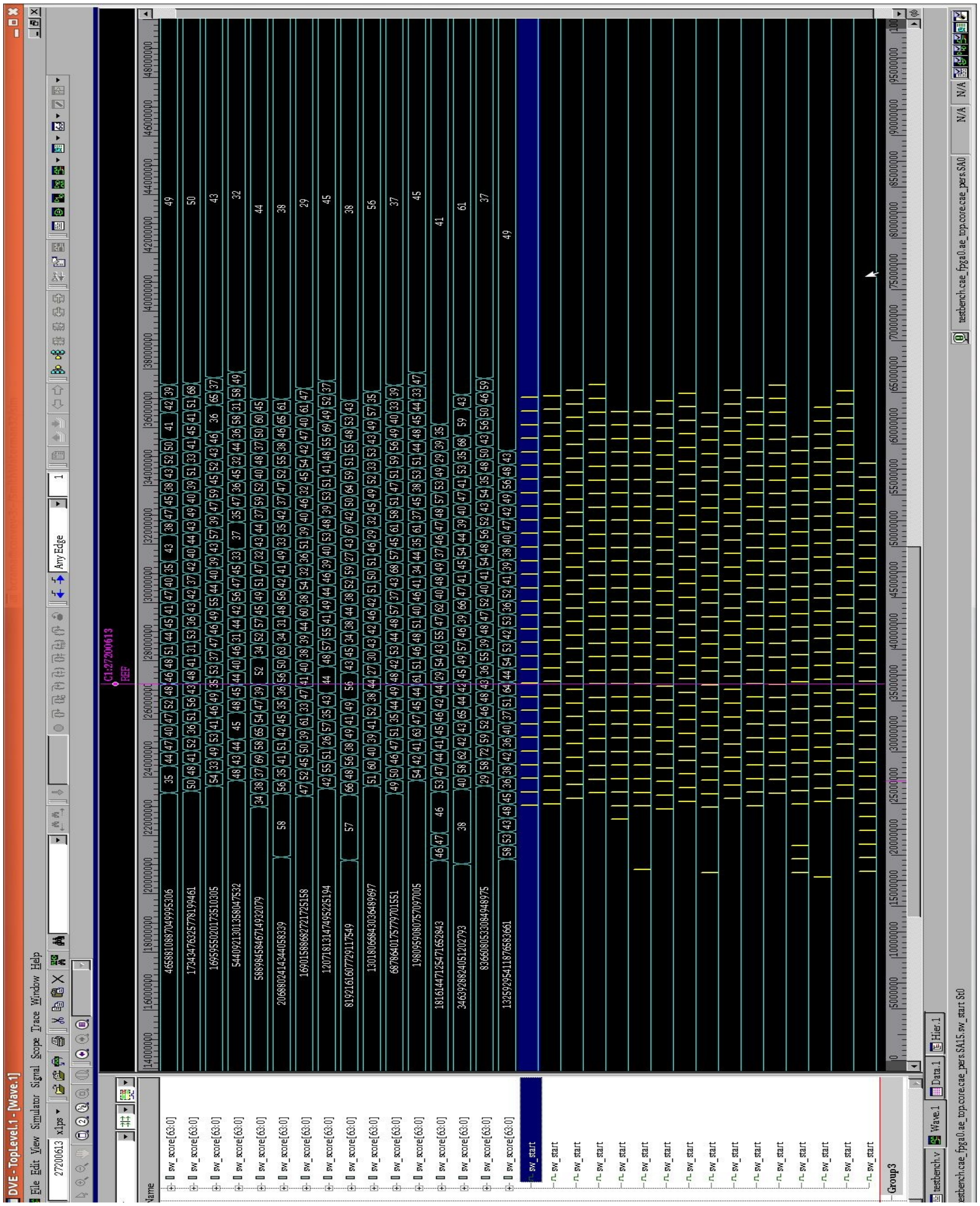
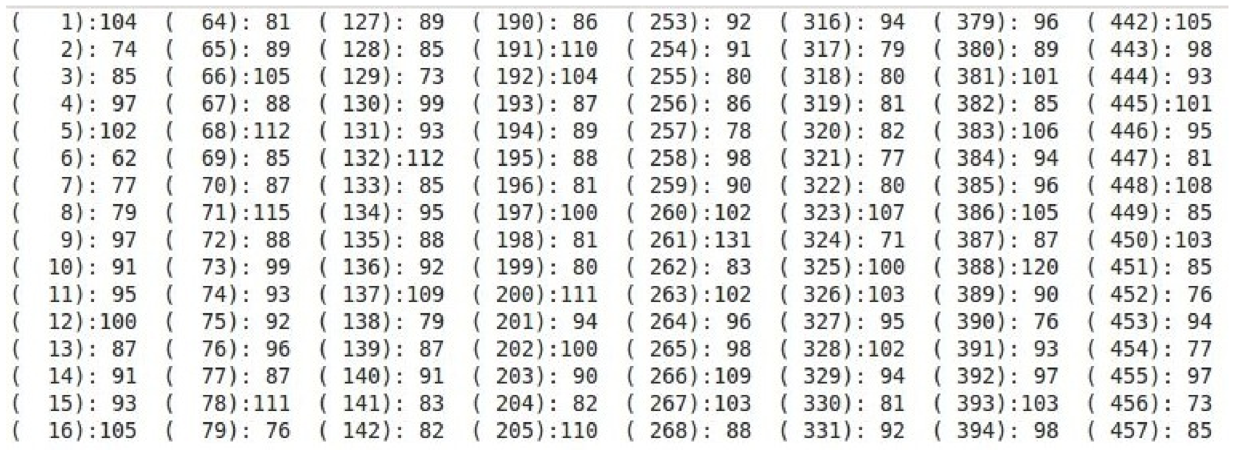

DVE - TopLevel.1 - [Wave.1]
File Edit View Simulator Signal Scope Trace Window Help
27200613 x1ps
Any Edge
C1:27200613
REF
Name
sw_score[63:0]
sw_start
Group3
testbench.v
Wave.1
Data.1
Hier.1
testbench.cae_fpga0.ae_top.core.cae_pers.SA15.sw_start S0
testbench.cae_fpga0.ae_top.core.cae_pers.SA0
N/A

Figure 5.3: Timing diagram showing the alignment score calculations.



```
(    1):104  (   64): 81  ( 127): 89  ( 190): 86  ( 253): 92  ( 316): 94  ( 379): 96  ( 442):105
(    2): 74  (   65): 89  ( 128): 85  ( 191):110  ( 254): 91  ( 317): 79  ( 380): 89  ( 443): 98
(    3): 85  (   66):105  ( 129): 73  ( 192):104  ( 255): 80  ( 318): 80  ( 381):101  ( 444): 93
(    4): 97  (   67): 88  ( 130): 99  ( 193): 87  ( 256): 86  ( 319): 81  ( 382): 85  ( 445):101
(    5):102  (   68):112  ( 131): 93  ( 194): 89  ( 257): 78  ( 320): 82  ( 383):106  ( 446): 95
(    6): 62  (   69): 85  ( 132):112  ( 195): 88  ( 258): 98  ( 321): 77  ( 384): 94  ( 447): 81
(    7): 77  (   70): 87  ( 133): 85  ( 196): 81  ( 259): 90  ( 322): 80  ( 385): 96  ( 448):108
(    8): 79  (   71):115  ( 134): 95  ( 197):100  ( 260):102  ( 323):107  ( 386):105  ( 449): 85
(    9): 97  (   72): 88  ( 135): 88  ( 198): 81  ( 261):131  ( 324): 71  ( 387): 87  ( 450):103
(   10): 91  (   73): 99  ( 136): 92  ( 199): 80  ( 262): 83  ( 325):100  ( 388):120  ( 451): 85
(   11): 95  (   74): 93  ( 137):109  ( 200):111  ( 263):102  ( 326):103  ( 389): 90  ( 452): 76
(   12):100  (   75): 92  ( 138): 79  ( 201): 94  ( 264): 96  ( 327): 95  ( 390): 76  ( 453): 94
(   13): 87  (   76): 96  ( 139): 87  ( 202):100  ( 265): 98  ( 328):102  ( 391): 93  ( 454): 77
(   14): 91  (   77): 87  ( 140): 91  ( 203): 90  ( 266):109  ( 329): 94  ( 392): 97  ( 455): 97
(   15): 93  (   78):111  ( 141): 83  ( 204): 82  ( 267):103  ( 330): 81  ( 393):103  ( 456): 73
(   16):105  (   79): 76  ( 142): 82  ( 205):110  ( 268): 88  ( 331): 92  ( 394): 98  ( 457): 85
```

Figure 5.4: The output of the accelerator: the sequence numbers with the scores of their pairwise alignment.

5.1.3 Simulation

The designs were simulated on Convey HDL simulator using the Synopsys tools. Figure 5.3 shows the timing diagram for the resultant scores being stored back to the memory. Every spike in the lower part of the diagram represents the availability at the compute cores and the start of an alignment. The steady and continuous flow of data and availability of results, represents a streamlined memory access. Thus, the assumptions made in the previous section are verified here.

Figure 5.4 shows the resultant scores generated from the simulation of 2000 sequences against another 2000 sequences. Each systolic array in the accelerator finds the pairwise alignment scores between two sequences and stores the scores in order in the co-processor memory. The scores were read from the co-processor memory and printed on to a file. These scores were verified from a software implementation and was tested for 100% accuracy in the values obtained.

5.2 Performance with GP-GPU

The GP-GPU programs were executed on two different devices, the NVIDIA GeForce GTX 580 and the NVIDIA GeForce GT 440. Both of these units are based on the NVIDIA Fermi Architecture [46]. However, some of the differences between them is highlighted in Table 5.1. Analysis were performed on the time taken for the program execution. The results and their analysis is discussed below.

The speedup is calculated by comparing the parallel version of the program on GPU over the serial version of the program running on a x86 processor with the following specifications:

Table 5.1: Key features of the platforms used

Features	**GT440**	**GTX580**
CUDA Capability	2.1	2.0
Total no of MPs	3	16
CUDA Cores/MP	48	32
GPU Clock Rate	1189 MHz	1544 MHz
Shared Memory per Block	49152 bytes	49152 bytes
Max Threads per Multiprocessor	1536	1536
Max Threads per Block	1024 bytes	1024
Registers Per Block	32768	32768

Model name : Intel(R) Core(TM) i7 CPU
CPU frequency : 960 @ 3.20GHz
Cache size : 8192 KB

The parallel version of the program was executed on GTX580 and Figure 5.5 shows the

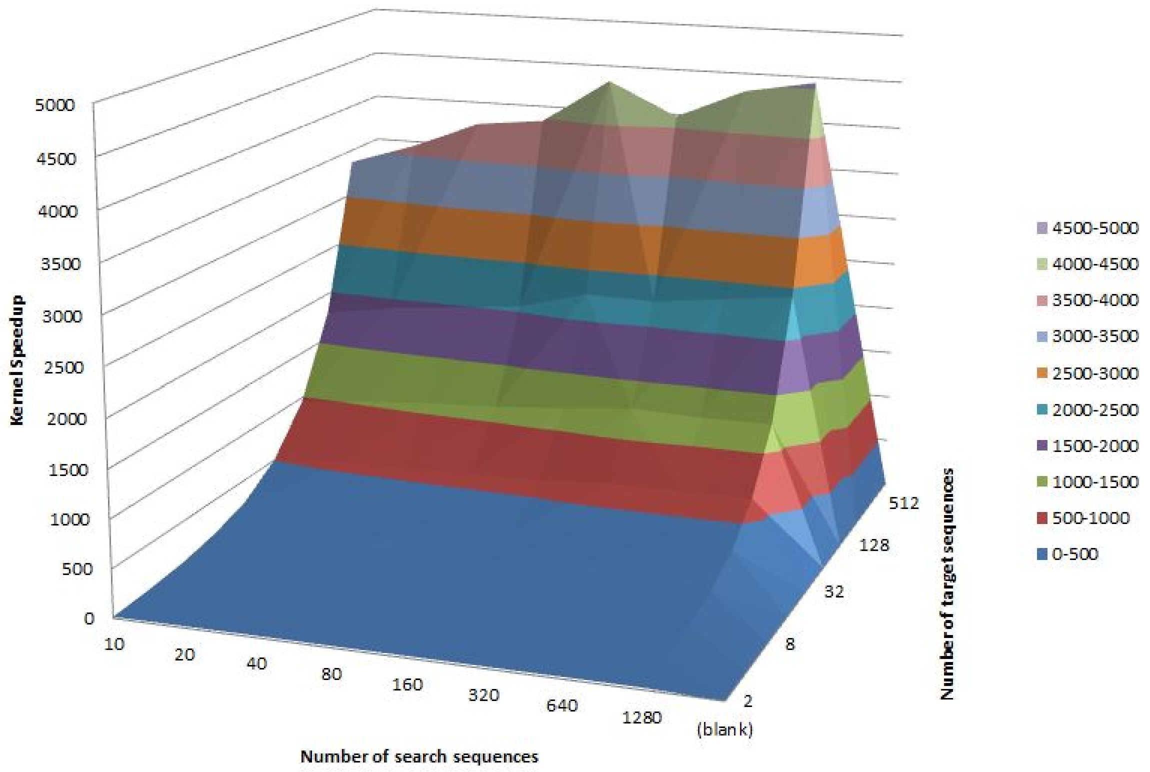


Figure 5.5: Kernel speedup with number of target and search sequences over x86.

raw kernel speedup attained with various query and reference database sizes over the serial x86 processor. Speedup of up to 4500x were observed as seen in Figure 5.5. The overhead of

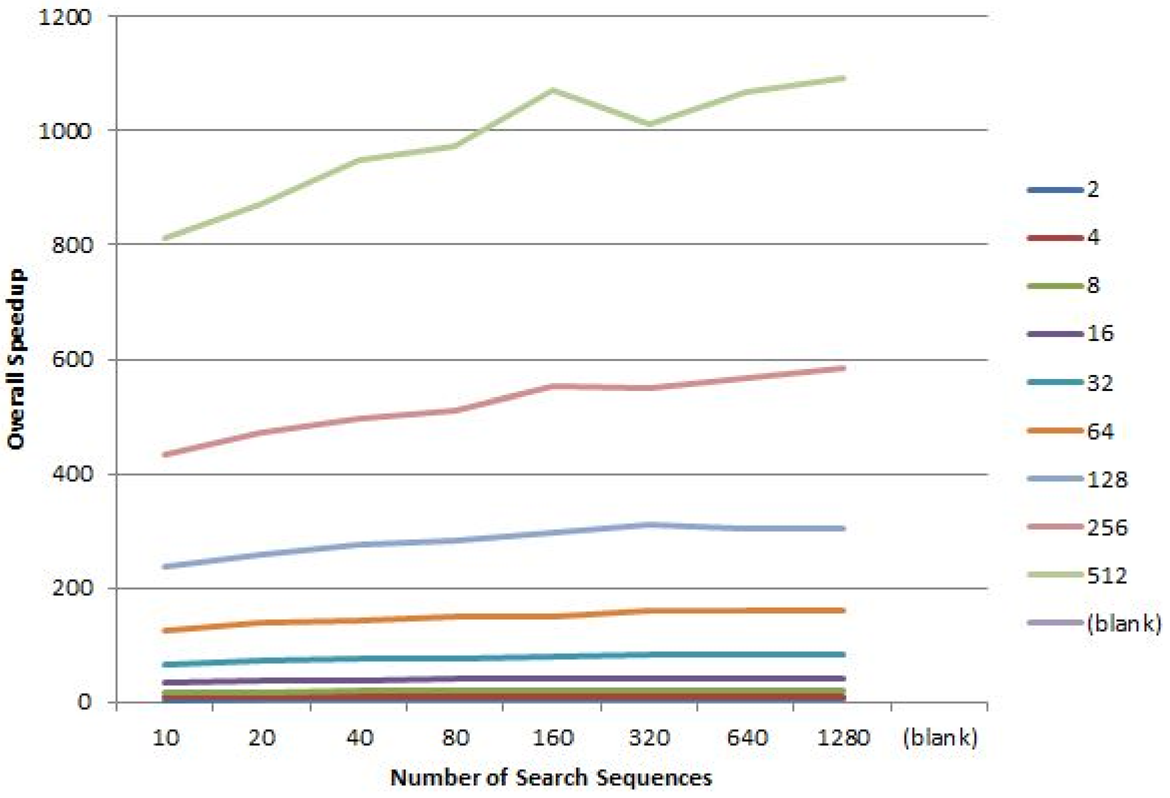


Figure 5.6: Overall speedup attained by using the GP-GPU over x86 processor.

transferring the data back and forth between the host and the device memory and the overhead of memory allocations on the GPU were considered while analyzing the overall trade off of using GPU parallel application. The graph plotted is seen in Figure 5.6. Speedup of up to 1100x have been attained for specific configurations of query and database sizes. This is partly due to the fact that in this approach, the score matrix is not involved in the memory copy overhead.

5.3 Performance Compared With Recent Work

This section compares the Smith-Waterman implemented in this work with some of the other recent contributions discussed in Section 2.7. The Table 5.2 compares the maximum throughput in GCUPS that can be attained with these implementations. Of the total sixteen implementations, seven of them are based on GP-GPUs, eight of them are based on FPGAs and one of them on Cell Broadband Engine. Of the seven GP-GPU based implementations, two of them are dual GPUs on a single graphics card.

Figure 5.7 shows the plot of these values on a bar chart. Logarithmic scale is used to accommodate the large scale ranging from 3.5 GCUPS on GeForce 8800 GP-GPU to 768 GCUPS in Convey HC-1ex with four Virtex-6 FPGAs. The red colored bars indicates the implementations from this work. It can be seen that Convey's implementation of their application *cnysws* comes close in performance to the Smith-Waterman implemented in this work. The graph however is dominated by the Convey's *cnysws* on HC-1ex with a throughput of 768 GCUPS followed by two other implementations on HC-1. The performance in the GTX 580 falls short of the other GP-GPU implementation, due to the lack of any changes to the native algorithm.

Table 5.2: Performance comparison with related work on heterogeneous computers.

Name	**Platform**	**Peak Performance (GCUPS)**
Issac et al. [32]	Altera Stratix EP1S40	23.8
Wirawan et al. [40]	Cell-Broadband Engine (BE)	3.6
Convey [3]	Convey HC-1	259
This Work	Convey HC-1	307.2
Convey [3]	Convey HC-1ex	768
CUDA++ [36]	NVIDIA GTX 295 (dual)	16.087
CUDA++3.0 [34]	NVIDIA GTX690 (dual)	185.6
Svetlin et al. [35]	NVIDIA GeForce 8800 GTX	3.5
This Work	NVIDIA GTX 580	9.98
CUDA Align 2.0 [38]	NVIDIA GTX285	23
CUDA++ [36]	NVIDIA GTX 280	9.66
CUDA++2.0 [15]	NVIDIA GTX 280	17
CUDA++3.0 [34]	NVIDIA GTX680	119
Altera white paper [25]	XtremeData XD1000	25.6
Allred et al. [33]	XtremeData IXD2000i	9.0
Oliver et al. [31]	Xilinx XC2V6000	5.0

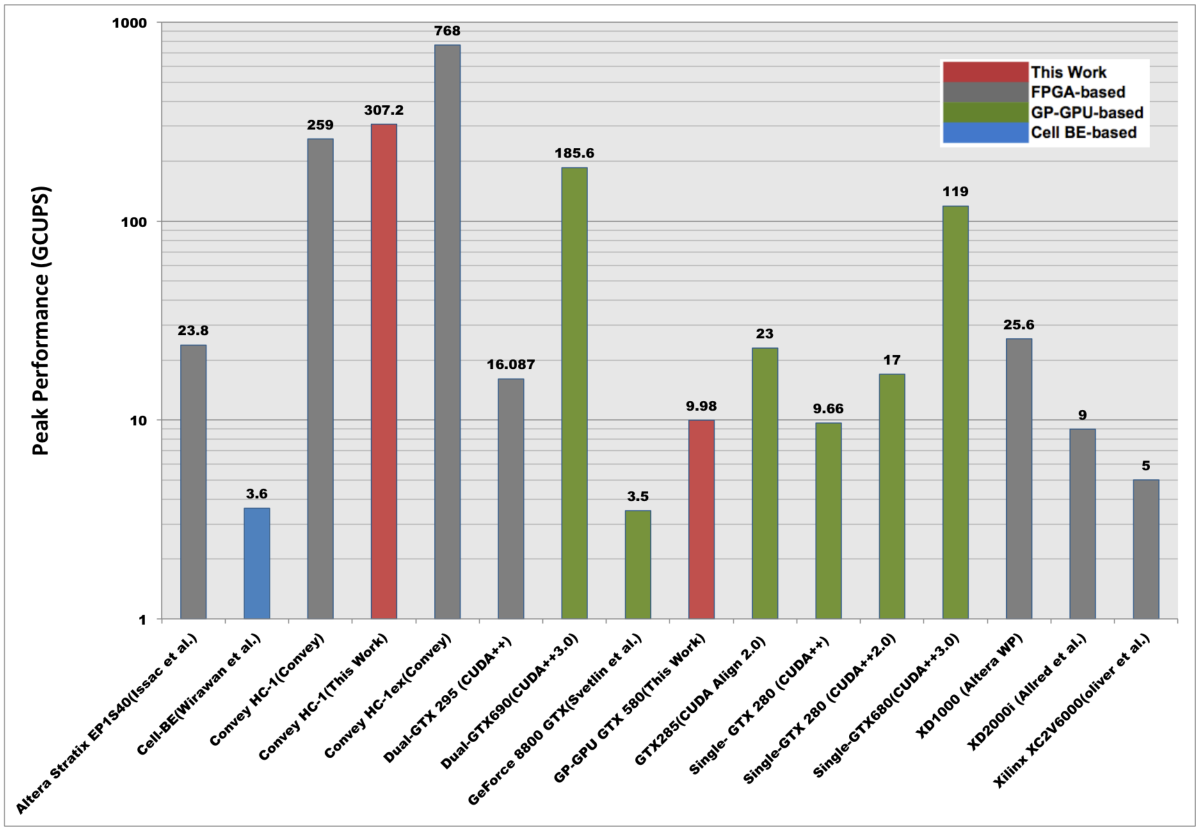


Figure 5.7: Throughput comparison of recent work on high performance Smith-Water Implementation.

5.4 LabVIEW Convey Interface

The LabVIEW Convey interface was successfully deployed with the ConVI design flow. A wrapper was built around the LabVIEW generated HDL, it was interfaced to the AEG registers and streamers successfully. The BPORT IN and the BPORT OUT were connected to the CSR and AEG registers. The updates from the VI front panel controls are passed through the ConVI execution server to the CSR/AEG resisters to update the values within the LabVIEW HDL. The values of the indicators were transferred from the LV HDL to the front panel.

ConVI as a design flow and LabVIEW as a HDL design tool was put to test during the VBI-NSF workshop. A team of five participants from different backgrounds were provided with a half-day training on NI LabVIEW. The team was provided with a basic Smith-Waterman LabVIEW implementation and was asked to make modifications to the design. The team was able to successfully come up with the modifications without any detailed knowledge of Convey PDK or HDL experience. Finally, the use of this tool for accelerator development received a positive response from the team.

The following subsections compare ConVI to other high level programming paradigms discussed in Section 2.7.

5.4.1 Comparing ConVI with Bflow

Design Environment: LabVIEW was primarily made for Virtual Instruments while Azido was specific to HDL design on Xilinx FPGA. Although Azido seems to have an advantage here, the LabVIEW design environment is more mature and stable and has a library of modules specially designed for FPGA implementations.

Design Files: ConVI runs on LabVIEW generated HDL while Bflow runs on Azido generated EDIF files. HDL files have the advantage where the source can be analyzed and can be modified. Wrappers can be easily designed for HDL files. EDIF files are netlists and they cannot be modified and does not have the flexibility that is present in HDL source files.

Interface Objects: Bflow connects to the hardware through COM objects while LabVIEW generates registers in hardware for every control and indicator. LabVIEW again has the advantage here since the register interface is independent of the design. This makes it easier to design the wrappers for the design as the interface is left unchanged and the LabVIEW communicates directly with the registers.

5.4.2 Comparison with OpenCL Smith-Waterman

OpenCL is another high level programming model that is used to program the different types of processors in an heterogeneous computer in a single programming paradigm. OpenCL was initially developed by Apple inc to support general purpose computing on GPUs across different vendors. This was later being extended to other platforms. Although OpenCL is a proven solution to program heterogeneous systems, the programmer still requires skill sets in parallel programming and must be architecture aware to deploy an application effectively.

LabVIEW offers the advantages of visualizing the modules and the parallelism in the algorithm. Architecture independent programming, real time monitoring and the ability to generate C code, places LabVIEW above OpenCL programming paradigm for FPGAs and FPGA based heterogeneous systems.

Chapter 6

Conclusion and Future Scope

This work demonstrated some of the ways of enabling the use of heterogeneous systems for bio-informatics applications. Here is an abridgment of the contributions from this work.

1. An efficient implementation of dynamic programming based Smith-Waterman was achieved. This work aimed to accelerate bio-applications by efficient usage of heterogeneous systems. The performance in GCUPS attained is higher than Convey's personality on HC-1.

2. A similar approach was used to implement Smith-Waterman on NVIDIA GP-GPUs using CUDA. This was compared to the performance of Smith-Waterman on Convey HC-1 and other GP-GPU/FPGA based systems.

3. The design of accelerator using LabVIEW was demonstrated. This proved to be a highly viable option to design and control the accelerator without of the domain knowledge and HDL skills. The ultimate aim of this work is to enable the bioinformaticians to be able to work free of hardware design engineers.

There is however many potential enhancements that can be incorporated in the future. The next three sections discusses the scope for future improvements in each of the tasks performed in this work.

6.1 Dynamic Programming Environment on Convey

With a dynamic programming personality being setup in Convey, one may use it for a wide variety of applications that have dynamic programming phase in them. BFAST and BWASW are the examples of two algorithms that accelerates the native Smith-Waterman. BFAST uses indexes to filter seeds while BWASW uses BWT to filter the number of alignment locations. Following is the description of the different phases of these two algorithms and how the Smith-Waterman hardware implementation can be incorporated. A similar approach can be followed for any algorithm with a Smith-Waterman phase in it.

6.1.1 Burrows-Wheeler Aligner Smith-Waterman (BWASW)

BWASW [5] involves the use of Burrows Wheelers Transform [47] to pre-process the genome sequences. A prefix tree is built from the reference sequence and a Prefix Dynamic Acyclic Word Graph (DAWG) is formed from the query sequence. The prefix DAWG is parsed over the prefix tree to find the alignment locations. The seeds are expanded using the Smith-Waterman algorithm making it suitable for long reads up to 100,000 base pair from Roche/454 sequencing technologies. The following are the steps in this algorithm,

1. Formation of Prefix trie of the reference sequence.
2. Formation of the Prefix-DAWG of the query sequence.
3. Aligning Prefix trie against Prefix DAWG.
4. Acceleration by the standard Smith-Waterman algorithm.

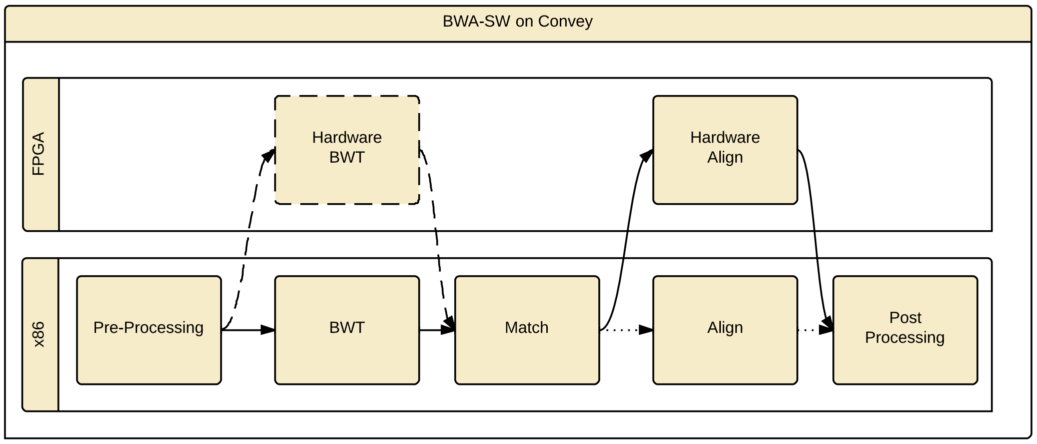


Figure 6.1: Proposed BWA-SW on Convey HC systems.

Figure 6.1 shows the proposed implementation of BWASW on Convey HC-1. As seen in the figure, the *Hardware Align* phase consists of Smith-Waterman which can be efficiently accelerated in the FPGA. The dotted line indicates a further enhancement to the applications. Depending on the load distribution, BWT phase can also be mapped to hardware [48] to optimize the performance.

6.1.2 BLAT like Fast and Accurate Search Tool (BFAST)

BFAST [7] aims to attain more accuracy by employing Smith-Waterman expansion. BFAST pre-computes an index of all non overlapping k-mers in the genome. This index fits inside the RAM of inexpensive computers, and need only be computed once for each genome assembly. It supports both Illumina and ABI's SOLiD reads. The following are the steps followed by BFAST,

1. Preprocessing the reference sequence.
2. Indexing process for the reference genome.
3. Matching the query with the indexed reference genome.
4. Local alignment using the Smith-Waterman algorithm.
5. Post processing and performing the alignments.

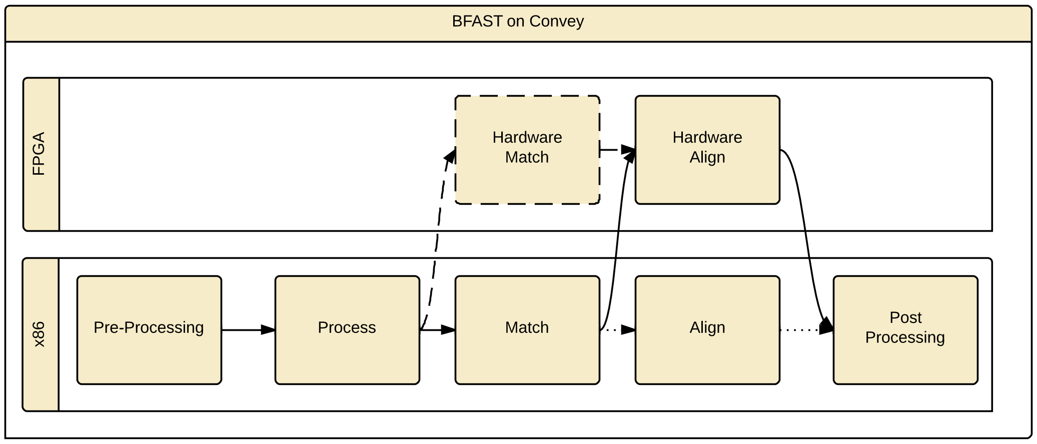


Figure 6.2: Proposed BFAST on Convey HC systems.

Figure 6.2 shows the proposed implementation of BFAST on Convey. The *Align* stage of BFAST can benefit from this Smith-Waterman implementation in the hardware. Depending on the parameters used, the match phase can be time consuming and would benefit from offloading it to hardware as demonstrated by Corey et al. [28].

6.2 Dynamic Parallelism using Kepler GP-GPUs

Dynamic Parallelism in CUDA is supported via an extension to the CUDA programming model that enables a CUDA kernel to create and synchronize new nested work [49]. Basically, a child CUDA kernel can be called from within a parent CUDA kernel and then optionally synchronize on the completion of that child CUDA kernel. The parent CUDA kernel can consume the output produced from the child CUDA kernel, all without the CPU involvement. Use of dynamic parallelism in Kepler architecture can further eliminate use of more threads by dynamically making decisions for launching threads within a kernel call. This would also enable the reuse of the scoring matrix.

6.3 Unified Programming with LabVIEW for Convey

Apart from the HDL generating capabilities of LabVIEW, it can create a C code that represents the algorithm developed graphically in LabVIEW [50]. With the LabVIEW C Generator, no target knowledge is required because the software generates ANSI C code from a LabVIEW application instead of generating target-specific binary code. This can be used to the advantage in generating the host code for Convey along with the hardware design. This enables the unification of the development platforms for both host and the FPGAs in Convey HC-1. Thus, LabVIEW has a great potential to be the development tool for programming heterogeneous platforms.

Bibliography

[1] *Convey HC Reference Manual.* Convey Computer, 2012.

[2] *Convey Personality Development Kit Reference Manual.* Convey Computer, 2012.

[3] *Smith-Waterman Datasheet.* Convey Computer, 2011.

[4] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283670900574

[5] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows Wheeler transform," vol. 26, no. 5, pp. 589–595, 2010.

[6] P. Gupta, "Swift: A GPU-based Smith-Waterman Sequence Alignment Program," May 2012.

[7] N. Homer, B. Merriman, and S. F. Nelson, "BFAST: an alignment tool for large scale genome resequencing." *PloS one*, vol. 4, no. 11, p. e7767, Jan. 2009. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2770639&tool=pmcentrez&rendertype=abstract

[8] "Isolating, Cloning, and Sequencing DNA," 2002. [Online]. Available: http://www.ncbi.nlm.nih.gov/books/NBK26837/

[9] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows Wheeler transform," vol. 25, no. 14, pp. 1754–1760, 2009.

[10] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*, 2nd ed. Cold Spring Harbor Press, 2004.

[11] A. Khalafallah, H. Elbabb, O. Mahmoud, and A. Elshamy, "Optimizing smith-waterman algorithm on graphics processing unit," in *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, 2010, pp. 650–654.

[12] S. Arming, R. Fenkhuber, and T. Handl, "Data Compression in Hardware The Burrows-Wheeler Approach," pp. 60–65, 2010.

[13] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, ser. FOCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 390–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795666.796543

[14] R. A. Baeza-Yates and G. H. Gonnet, "Fast text searching for regular expressions or automaton searching on tries," *J. ACM*, vol. 43, no. 6, pp. 915–936, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/235809.235810

[15] Y. Liu, B. Schmidt, and D. Maskell, "Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," *BMC Research Notes*, vol. 3, no. 1, p. 93, 2010. [Online]. Available: http://www.biomedcentral.com/1756-0500/3/93

[16] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283681900875

[17] "FPGAs," http://www.altera.com/products/fpga.html.

[18] *Virtex-5 Family Overview*. Xilinx, february 2009.

[19] D. Uliana, K. Kepa, and P. Athanas, "Fpga-based hpc application design for non-experts (abstract only)," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 274–274. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435334

[20] Z. Jin and J. D. Bakos, "Extending the BEAGLE library to a multi-FPGA platform." *BMC bioinformatics*, vol. 14, no. 1, p. 25, Jan. 2013. [Online]. Available: http://www.biomedcentral.com/1471-2105/14/25

[21] *Convey Coprocessor Memory Ordering*. Convey Computer, january 2012. [Online]. Available: http://www.conveysupport.com/alldocs/WhitePapers/PDK/WP_Coprocessor_Memory_Order.pdf

[22] *LabVIEW 2012 FPGA Module Help*. National Instruments.

[23] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann, 2010.

[24] *NVIDIA CUDA Programming Guide 2.0*. NVIDIA, 2008.

[25] "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," *ALTERA Whitepaper*, 2007.

[26] Y. Chen, B. Schmidt, and D. L. Maskell, "Accelerating short read mapping on an fpga (abstract only)," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 265–265. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145740

[27] E. Fernandez, W. Najjar, and S. Lonardi, "String matching in hardware using the fm-index," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 218–225. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2011.55

[28] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware Acceleration of Short Read Mapping," *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 161–168, Apr. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6239809

[29] S. I. Steinfadt, "SWAMP+: Enhanced Smith-Waterman Search for Parallel Models," in *2012 41st International Conference on Parallel Processing Workshops*. IEEE, Sep. 2012, pp. 62–70. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6337464

[30] G. Chrysos, E. Sotiriades, C. Rousopoulos, A. Dollas, A. Papadopoulos, I. Kirmitzoglou, V. Promponas, T. Theocharides, G. Petihakis, J. Lagnel, P. Vavylis, and G. Kotoulas, "Opportunities from the use of fpgas as platforms for bioinformatics algorithms," in *Bioinformatics Bioengineering (BIBE), 2012 IEEE 12th International Conference on*, 2012, pp. 559–565.

[31] T. Oliver, B. Schmidt, and D. Maskell, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 12, pp. 851–855, Dec. 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1556805

[32] I. T. S. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)." *BMC bioinformatics*, vol. 8, no. 1, p. 185, Jan. 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/185

[33] J. Allred, J. Coyne, W. Lynch, V. Natoli, J. Grecco, and J. Morrissette, "Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, May 2009, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5161214

[34] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," *BMC Bioinformatics*, vol. 14, no. 1, p. 117, 2013. [Online]. Available: http://www.biomedcentral.com/1471-2105/14/117

[35] S. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/S2/S10

[36] Y. Liu, D. Maskell, and B. Schmidt, "Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units," *BMC Research Notes*, vol. 2, no. 1, p. 73, 2009. [Online]. Available: http://www.biomedcentral.com/1756-0500/2/73

[37] E. F. O. Sandes and A. C. M. de Melo, "Cudalign: using gpu to accelerate the comparison of megabase genomic sequences," *SIGPLAN Not.*, vol. 45, no. 5, pp. 137–146, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693473

[38] E. de O Sandes and A. de Melo, "Smith-waterman alignment of huge sequences with gpu in linear space," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 1199–1211.

[39] J. Li, S. Ranka, and S. Sahni, "Pairwise sequence alignment for very long sequences on gpus," in *Computational Advances in Bio and Medical Sciences (ICCABS), 2012 IEEE 2nd International Conference on*, 2012, pp. 1–6.

[40] A. Wirawan, C. K. Kwoh, N. T. Hieu, and B. Schmidt, "CBESW: sequence alignment on the Playstation 3." *BMC bioinformatics*, vol. 9, no. 1, p. 377, Jan. 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/377

[41] A. M. Aji, "Exploiting Multigrain Parallelism in Pairwise Sequence Search on Emergent CMP Architectures," Aug. 2008. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-06162008-171147/

[42] K. S. Shagrithaya, "Enabling Development of OpenCL Applications on FPGA platforms," Sep. 2012. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-08202012-095025/

[43] K. Kepa, R. Soni, and P. Athanas, "OpenCL2FPGA," Dec. 2012.

[44] "Ni labs: Labview vi scripting," 2012. [Online]. Available: http://sine.ni.com/nips/cds/view/p/lang/en/nid/209110

[45] *Virtex-6 Family Overview.* Xilinx, january 2012.

[46] *FERMI Compute Architecture White Paper.* NVIDIA, 2009.

[47] M. Burrows and D. Wheeler, “A block-sorting lossless data compression algorithm,” 1994. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.5254

[48] J. Martínez, R. Cumplido, C. Feregrino, L. Enrique, E. Sta, and M. Tonantzintla, “An FPGA-based Parallel Sorting Architecture for the Burrows Wheeler Transform,” no. ReConFig, pp. 0–6, 2005.

[49] *NVIDIA CUDA Dynamic Parallelism Programming Guide.* NVIDIA, 2012.

[50] *LabVIEW C Code Generation Technology Basics.* National Instruments, Dec. 2010. [Online]. Available: http://www.ni.com/white-paper/11784/en/

Appendix A

Smith-Waterman State Machine

```
//  State machine for Smith Waterman
   always @*
   begin
   case(sw_pstate)
   INITIAL:
   begin             // Initial state to begin at every reset
     sw_hold = 1'b1;       // the pipe is on hold
          push_result = 1'b0;       // the fifo are not used
          pop_fifo = 1'b0;
     sw_enable = 1'b0;
          if(c_result_vld)           // when the valid signal is seen , FSM goes
    to IDLE
       begin
       sw_nstate = IDLE;
       pop_fifo = 1'b1;
       end
     else
       sw_nstate = INITIAL;
          end

   IDLE:                     // Default state ar every wait time
     begin
     sw_hold = 1'b1;         // the pipe is on hold
          push_result = 1'b0;     // the fifo are not used
          pop_fifo = 1'b0;
          if(c_result_vld)        // When both the fifo have the data
       begin
       sw_nstate = LD_PIPE;   // go to load the pipe
       end
     else
       sw_nstate = IDLE;
          end

        LD_PIPE:                  // state to load the pipes with data
     begin
     sw_enable = 1'b1;        // the pipes are enabled
          pop_fifo = 1'b1;          // the value from the fifo are popped out
     sw_nstate = START;          //go to start of processing
     end

        START:
     begin
          pop_fifo = 1'b0;
          sw_hold  = 1'b0;
     sw_enable = 1'b1;
     if (sw_count > 6'd32)     // stay in start until all the data is pushed
    into the pipes
      sw_nstate = STOP;     // move to stop after all data is streamed into the
    pipes
     else
      sw_nstate = START;
          end

   STOP:
     begin
     sw_enable = 1'b0 ;       // Set enable to zero to disable the cells in the
    pipes one by one
          if (pipe_done)           // Wait until the computation is done
              begin
         sw_hold  = 1'b1;      // hold the values
         sw_score[63:0] = {49'b0,pipe_score[UMAX:0]}; // get the maximum score
    from the pipe
              sw_nstate = STORE;                         // store the score
    into the fifo
         end
     else
         begin
              sw_nstate = STOP;
         end
          end

   STORE:                           // four
       begin
       r_result[63:0] = sw_score[63:0];             // copy the result to the
    fifo register
```

```
            push_result = 1’b1;           // push the result into the fifl
            sw_nstate = IDLE;          // go back to idle
       end

     default:  sw_nstate = IDLE;
  endcase
end
```

Appendix B

Smith-Waterman PE and Pipeline Modules

```
// Systolic Array: Cascaded Smith-Waterman Processing Elements
genvar i;
generate
   for (i = 0; i < NUM_CELLS; i = i + 1)
      begin:SW_Cell

         // SW Cell
         sw_cell swc (
            .t_in   (pipe_t_in[i]),         // target sequence in from
 previous cell
            .max_in (pipe_max_in[i]),     // maximum in from  previous cell
            .v_in   (pipe_v_in[i]),         // cell value in from previous
 cell
            .st_in  (pipe_st_in[i]),       // search sequence in from
 previous cell
            .clr_in (pipe_clr_in[i]),     // clear signal in from previous
 cell
            .hold_in(pipe_hold_in[i]),          // hold signal in from
 previous cell
            .s_in   (s),
            .clk    (clk),                    // clock to the cell
            .t_out  (pipe_t_in[i+1]),      // target out to the next cell
            .max_out(pipe_max_in[i+1]),  // max out to the next cell
            .v_out  (pipe_v_in[i+1]),      // score out to the next cell
            .st_out (pipe_st_in[i+1]),    // search sequence out to the next
 cell
            .clr_out(pipe_clr_in[i+1]),   // clear out to the next cell
```

```
              .hold_out(pipe_hold_in[i+1])          // hold signal in from
   previous cell
           );

        end
   endgenerate

//--------------------------------------------------------------

   // Smith-Waterman Pipeline
      sw_pipe swp(
      .t(pipe_t), // operand 1
      .s(pipe_s), // operand 2
      .en(sw_enable),      // enable signal
      .hold(sw_hold),      // hold signal
      .start(sw_start),    // start signal
      .clk(clk),           // clock signal
      .score(pipe_score), // operand 3
      .done(pipe_done)    // pipe done signal
      );
      defparam swp.NUM_CELLS = NUM_CELLS;
      defparam swp.SMAX      = SMAX;
           defparam swp.UMAX      = UMAX;
      defparam swp.GAP_OPEN  = GAP_OPEN;
      defparam swp.GAP_EXT   = GAP_EXT;
//--------------------------------------------------------------
```

Appendix C

Smith-Waterman Host Code

```
//--------------------------------------------------------------------------------
// Institute: Virginia Tech
// Author: Ramakrishna
//--------------------------------------------------------------------------------
#include <convey/usr/cny_comp.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#undef DEBUG

typedef unsigned long long uint64;
extern long cpSWM();
extern int cpXbar();
void usage (char *);
void load_memory(uint64 *a1, long  , char* );

int main(int argc, char *argv[])
{
  long i;
  long size;
  uint64  *a1, *a2, *a3;
  uint64 act_sum;
  uint64 exp_sum=0;
  int index;
  char* filename = "scores.txt";
  FILE *output_file;

  // check command line args
  if (argc == 1)
```

```c
    size = 2000;     // default size
  else if (argc == 2)
    {
    size = atoi(argv[1]);
    if (size > 0)
    {
      printf("Running SmithWaterman.exe with size = %lld\n", size);
    }
    else
    {
      usage (argv[0]);
      return 0;
    }
  }
  else
    {
    usage (argv[0]);
    return 0;
    }

  // Get personality signature
  cny_image_t        sig2;
  cny_image_t        sig;
  int stat;
  if (cny_get_signature)
    cny_get_signature("pdk", &sig, &sig2, &stat);
  else
    fprintf(stderr,"ERROR:  cny_get_signature not found\n");

  if (stat) {
    printf("***ERROR: cny_get_signature() Failure: %d\n", stat);
    exit(1);
  }

  // check interleave
  // this example requires binary interleave
  if (cny_cp_interleave() == CNY_MI_3131) {
    printf("ERROR - interleave set to 3131, this personality requires binary
   interleave\n");
    exit (1);
  }

  // read AEG control register to see if crossbar is present
  int crossbar_enabled = 0;
  crossbar_enabled = i_copcall_fmt(sig, cpXbar, "");
```

```
#ifdef DEBUG
  printf("UserApp:  crossbar_enabled = %d\n", crossbar_enabled);
#endif

  // Allocate memory
  // if the crossbar is enabled in the hardware, use malloc
  // if no crossbar, the memory must be aligned on MC 0
  if (crossbar_enabled)  {
      if (cny_cp_malloc)  {
   a1 = (uint64 *) (cny_cp_malloc)(size*8);
   a2 = (uint64 *) (cny_cp_malloc)(size*8);
   a3 = (uint64 *) (cny_cp_malloc)(size*8);
      }
      else
   printf("malloc failed\n");
  }
  else {
   cny_cp_posix_memalign((void**)&a1, 512, size*8);
   cny_cp_posix_memalign((void**)&a2, 512, size*8);
   cny_cp_posix_memalign((void**)&a3, 512, size*8);
  }

#ifdef DEBUG
  printf("a1 = %llx MC = %lld \n", a1, ((uint64)a1>>6)&7);
  printf("a2 = %llx MC = %lld \n", a2, ((uint64)a2>>6)&7);
  printf("a3 = %llx MC = %lld \n", a3, ((uint64)a3>>6)&7);
#endif

  // populate operand arrays with sequences, initialize a3
 load_memory(a1, size , "sequences1.fasta");
 load_memory(a2, size , "sequences2.fasta");

  // Smith-Waterman copcall
  act_sum = l_copcall_fmt(sig, cpSWM, "AAAA", a1, a2, a3, size);
   output_file = fopen(filename, "w+"); // write mode

   for ( index =0; index < size ; index+=4)
        fprintf(output_file,"\na3[%d]= %lld,%lld,%lld,%lld",index,
    a3[index],a3[index+1],a3[index+2],a3[index+4]);
   fclose(output_file);

   for ( index =0; index < 20 ; index++)
        printf("\na1[%d]= %lld     a2[%d]=%lld",index,
    a1[index],index,a2[index]);

  return 0;
}

// Print usage message and exit with error.
void
usage (char* p)
{
    printf("usage: %s [count (default 100)] \n", p);
    exit (1);
}

void load_memory(uint64 *a1, long size , char* filename)
{
char ch[80];
FILE *in;
long i;
int n=0,m=0;
unsigned long zero = 0 ,one = 1 ,two = 2 ,three = 3;

   in = fopen(filename,"r+"); // read mode

   if( in == NULL )
   {
      perror("Error while opening the file.\n");
      exit(EXIT_FAILURE);
   }

   for(n=0;fgets(ch,80,in) && n < size;)
   {
     if((ch[0] != 'a') && (ch[0] != 'g') && (ch[0] != 't') && (ch[0] != 'c'))
     continue ;

      a1[n] = 0;
      for( i=0 ; i<32; i++)
      {
         if       (ch[i] == 'a') a1[n] = a1[n]|(zero<<(2*i));
         else  if (ch[i] == 't') a1[n] = a1[n]|(one<<(2*i));
         else  if (ch[i] == 'g') a1[n] = a1[n]|(two<<(2*i));
         else  if (ch[i] == 'c') a1[n] = a1[n]|(three<<(2*i));
         }
     n++;
   }
}
```

Appendix D

LabVIEW HDL Connection

```
 assign aeg_flag = aeg_lv_in[60];
 always @(posedge clk) begin

       new_aeg_id [1:0]  <=     aeg_lv_in[57:56];
       old_aeg_id [1:0]  <=     new_aeg_id [1:0];
       new_csr_id [1:0]  <=     csr_lv_in[57:56];
       old_csr_id [1:0]  <=     new_csr_id [1:0];

 if (new_aeg_id != old_aeg_id) begin
       data_in[31:0]  <=  aeg_lv_in[31:0];
       rd_in        <=  aeg_lv_in[48];
       wr_in        <=  aeg_lv_in[52];
       address_in[10:0]  <=  aeg_lv_in[42:32];
       ret_id            <=       new_aeg_id;
 end
 else if (new_csr_id != old_csr_id) begin

       data_in[31:0]  <=  csr_lv_in[31:0];
       rd_in        <=  csr_lv_in[48];
       wr_in        <=  csr_lv_in[52];
       address_in[10:0]  <=  csr_lv_in[42:32];
       ret_id            <=       new_csr_id;
 end
 else begin
       rd_in        <=  1'b0;
       wr_in        <=  1'b0;
       end
 end

always @(posedge clk) begin
```

```
    if (valid) begin
     csr_lv_out[31:0]   <=    data_out[31:0];
     csr_lv_out[57:56]  <=    ret_id[1:0];
    end
    csr_lv_out[48]     <=    ready;
    csr_lv_out[52]     <=    valid;

 end

```