Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs

Tim Oliver, Bertil Schmidt and Douglas Maskell
Centre for High Performance Embedded Systems
Nanyang Technological University
Singapore
tim.oliver@pmail.ntu.edu.sg

ABSTRACT
Protein sequences with unknown functionality are often compared to a set of known sequences to detect functional similarities. Efficient dynamic-programming algorithms exist for solving this problem, however current solutions still require significant scan times. These scan time requirements are likely to become even more severe due to exponential database growth. In this paper we present a new approach to bio-sequence database scanning using re-configurable FPGA-based hardware platforms to gain high performance at low cost. Efficient mappings of the Smith-Waterman algorithm using fine-grained parallel processing elements (PEs) that are tailored towards the parameters of a query have been designed. We use customization opportunities available at run-time to dynamically hyper customize the systolic array to make better use of available resource. Our FPGA implementation achieves a speedup of approximately 170 for linear gap penalties and 125 for affine gap penalties as compared to a standard desktop computing platform. We show how hyper-customization at run-time can be used to further improve the performance.

Categories and Subject Descriptors
B.2.4 [**High-Speed Arithmetic**]: Cost/performance C.3 [**Special-Purpose And Application-Based Systems**]: Signal processing systems; J.3 [**Life And Medical Sciences**]: Biology and genetics

General Terms
Algorithms, Design, Performance, Experimentation.

Keywords
FPGA, Dynamic Re-configuration, Smith-Waterman, Bio-informatics

FPGA'05, February 20–22, 2005, Monterey, California, USA.

1. INTRODUCTION
Scanning protein sequence databases is a common and often repeated task in molecular biology. The need for speeding up this treatment comes from the exponential growth of the bio-sequence banks: every year their size scales by a factor of 1.5 to 2. The scan operation consists of finding similarities between a particular query sequence and all sequences of a bank. This operation allows biologists to point out sequences sharing common subsequences. From a biological point of view, it leads to identifying similar functionality.

Comparison algorithms whose complexities are quadratic with respect to the length of the sequences detect similarities between the query sequence and a subject sequence. One frequently used approach to speed up this time consuming operation is to introduce heuristics in the search algorithm [1]. The main drawback of this solution is that the more time efficient the heuristics, the worse is the quality of the result [18].

Another approach to get high quality results in a short time is to use parallel processing. There are two basic methods of mapping the scanning of sequence databases to a parallel processor: one is based on the systolisation of the sequence comparison algorithm; the other is based on the distribution of the computation of pair-wise comparisons. Systolic array architectures have been proven as a good candidate structure for the first approach [6],[13],[20], while more expensive supercomputers and networks of workstations are suitable architectures for the second [8],[16].

Special-purpose systolic arrays provide the best area/performance ratio by means of running a particular algorithm [15]. Their disadvantage is the lack of flexibility with respect to the implementation of different algorithms. Several massively parallel SIMD architectures have been developed in order to combine the speed and simplicity of systolic arrays with flexible programmability [4],[7],[21]. However, because of the high production costs involved, there are many cases where announced second-generation architectures have not been produced. The strategy to high performance sequence database scanning used in this paper is based on FPGAs, which provide a flexible platform for fine-grained parallel computing based on re-configurable hardware. Since there is a large overall FPGA market, this approach has a relatively small price/unit and also facilitates upgrading to FPGAs based on state-of-the-art technology. Taking full advantage of hardware reconfiguration, we present PE designs that are tailored towards particular query parameters. We will show how this leads to a high-speed implementation on a Virtex II XC2V6000. The implementation is also portable to other FPGAs.

This paper is organized as follows. In Section 2, we introduce the basic sequence comparison algorithm for database scanning. Section 3 highlights previous work in parallel sequence comparison. The parallel algorithm and its mapping onto the FPGA platform are explained in Section 4. The performance is evaluated and compared to previous implementations in Section 5. Dynamic re-configuration is considered in section 6. Section 7 concludes the paper.

2. SEQUENCE COMPARISON

Surprising relationships have been discovered between protein sequences that have little overall similarity but in which similar subsequences can be found. In that sense, the identification of similar subsequences is probably the most useful and practical method for comparing two sequences. The Smith-Waterman algorithm [22] finds the most similar subsequences of two sequences (the local alignment) by dynamic programming.

The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through a series of such elementary operations, any segment can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken as the measure of the distance between the segments.

Consider two strings $S1$ and $S2$ of length $l1$ and $l2$. To identify common subsequences, the Smith-Waterman algorithm computes the similarity $H(i,j)$ of two sequences ending at position i and j of the two sequences $S1$ and $S2$. The computation of $H(i,j)$ is given by the following recurrences:

$$H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1)+Sbt(S1_i,S2_j)\},$$

for $1 \le i \le l1$, $1 \le j \le l2$.

$$E(i,j) = \max\{H(i,j-1)-\alpha, E(i,j-1)-\beta\}, \quad 0 \le i \le l1,\ 1 \le j \le l2.$$

$$F(i,j) = \max\{H(i-1,j)-\alpha, F(i-1,j)-\beta\}, \quad 1 \le i \le l1,\ 0 \le j \le l2.$$

where Sbt is a character substitution cost table. Initialization of these values are given by $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \le i \le l1$, $0 \le j \le l2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gaps. This type of gap cost is known as *affine gap penalty*. Some applications also use a *linear gap penalty*, i.e. $\alpha = \beta$. For linear gap penalties the above recurrence relations can be simplified to:

$$H(i,j) = \max\{0, H(i,j-1)-\alpha, H(i-1,j)-\alpha, H(i-1,j-1) + Sbt(S1_i,S2_j)\},$$

for $1 \le i \le l1$, $1 \le j \le l2$.

$$H(i,0) = H(0,j) = 0 \text{ for } 0 \le i \le l1,\ 0 \le j \le l2.$$

Each position of the matrix H is a similarity value. The two segments of $S1$ and $S2$ producing this value can be determined by a backtracking procedure.

Figure 1 shows an example of the Smith-Waterman algorithm used to compute the local alignment between two DNA sequences ATCTCGTATGAT and GTCTATCAC. The matrix $H(i,j)$ is shown for the linear gap cost $\alpha = 1$, and a substitution cost of +2 if the characters are identical and −1 otherwise. From the highest score (+10 in the example), a trace-back procedure delivers the corresponding alignment (shaded cells), the two subsequences TCGTATGA and TCTATCA.

	∅	A	T	C	T	C	G	T	A	T	G	A	T
∅	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	1	0	0	2	1	0
T	0	0	2	1	2	1	1	4	3	2	1	1	3
C	0	0	1	4	3	4	3	3	3	2	1	0	2
T	0	0	2	3	6	5	4	5	4	5	4	3	2
A	0	2	2	2	5	5	4	4	7	6	5	6	5
T	0	1	4	3	4	4	4	6	5	9	8	7	8
C	0	0	3	6	5	6	5	5	5	8	8	7	7
A	0	2	2	5	5	5	5	4	7	7	7	10	9
C	0	1	1	4	4	7	6	5	6	6	6	9	9

Figure 1. Example of the Smith-Waterman algorithm

3. PREVIOUS WORK

A number of parallel architectures have been developed for sequence analysis. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving the sequence alignment problems.

Special-purpose hardware implementations can provide the fastest means of running a particular algorithm with very high PE density. However, they are limited to one single algorithm, and thus cannot supply the flexibility necessary to run a variety of algorithms required analyzing for DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [17]. More recent examples, better tuned to the needs of computational biology, include BISP, SAMBA, and BIOSCAN [6],[13],[20].

An approach presented in [21] is based on instruction systolic arrays (ISAs). ISAs combine the speed and simplicity of systolic arrays with flexible programmability. Several other approaches are based on the SIMD concept, e.g. MGAP [4], Kestrel [7], and Fuzion [21]. SIMD and ISA architectures are programmable and can be used for a wider range of applications, such as image processing and scientific computing. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose ASICs. Nevertheless, SIMD solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

Re-configurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs) or custom-designed arrays. They are generally slower and have lower PE densities than special-purpose architectures. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. Several solutions including Splash-2 [14] and Decypher [24] are based on FPGAs while PIM has its own re-configurable design [9]. Solutions based on FPGAs have the additional advantage that they can be regularly upgraded to state-of-the-art-technology. This makes FPGAs a very attractive alternative to special-purpose and SIMD architectures.

Compared to the previously published FPGA solutions, we are using a new partitioning technique for varying query sequence

lengths. The design presented in [25] is closest to our approach since it also uses a linear array of PEs on a re-configurable platform. Unfortunately, it only allows for linear gap penalties and global alignment, while our implementation considers both linear and affine gap penalties and is able to compute local alignments.

4. MAPPING ON A FPGA PLATFORM

The dynamic programming calculation presented in Section 2 can be efficiently mapped to a linear array of PEs. A common mapping is to assign one PE to each character of the query string, and then to shift a subject sequence systolically through the linear chain of PEs. In Figure 2 the query sequence is loaded into the processor array (one character per PE) and a subject sequence flows from left to right through the array. During each step, one elementary matrix computation is performed in each PE. If M is the length of the first sequence and K is the length of the second, the comparison is performed in $M+K-1$ steps on M PEs, instead of $M \times K$ steps required on a sequential processor. In each step the computation for dynamic programming cells along a single diagonal in Figure 1 is performed in parallel.

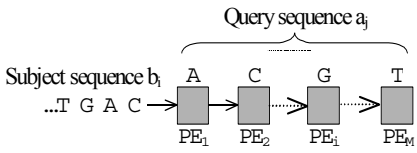

Figure 2. Sequence comparison on a linear processor array

Taking advantage of having a reconfigurable hardware platform, we can tailor the individual PE design towards different gap penalty functions. This approach allows us to include only as much computational hardware and local memory as required. Figure 3 shows our designs for linear gap penalties and for affine gap penalties. Data width (*dw*) is scaled to the required precision (usually *dw*=16 is sufficient). The LUT depth is scaled to hold the required number of substitution table rows. Substitution width (*sw*) is scaled to accommodate the dynamic range required by the substitution table. Look-up address width (*lw*) is scaled in relation to the LUT depth. Each PE has local memory to store $H(i,j-1)$, $H(i-1,j)$ and $H(i-1,j-1)$. The PE holds a column of the substitution table in its LUT. The look-up of $Sbt(a_i,b_j)$ and addition to $H(i-1,j-1)$ is done in one cycle. The score is calculated in the next cycle and passed to the next PE in the array. The PE keeps track of the maximum score computed so far and passes it to the next PE in the array. The affine gap penalty PE has additional storage for $E(i-1,j)$ and $F(i,j-1)$ and additional score computation circuitry. Additions are performed using saturation arithmetic.

Assuming, we are aligning the sequences $A = a_1a_2\ldots a_M$ and $B = b_1b_2\ldots b_K$, on a linear processor array of size M with affine gap penalties, where A is the query sequence and B is a subject sequence of the database. As a preprocessing step, symbol a_i, is assigned to PE i, $1 \le i \le M$. After that the row of the substitution table corresponding to the respective character is loaded into each PE as well as the gap penalties α and β. B is then completely shifted through the array in $M+K-1$ steps as displayed in Figure 2.

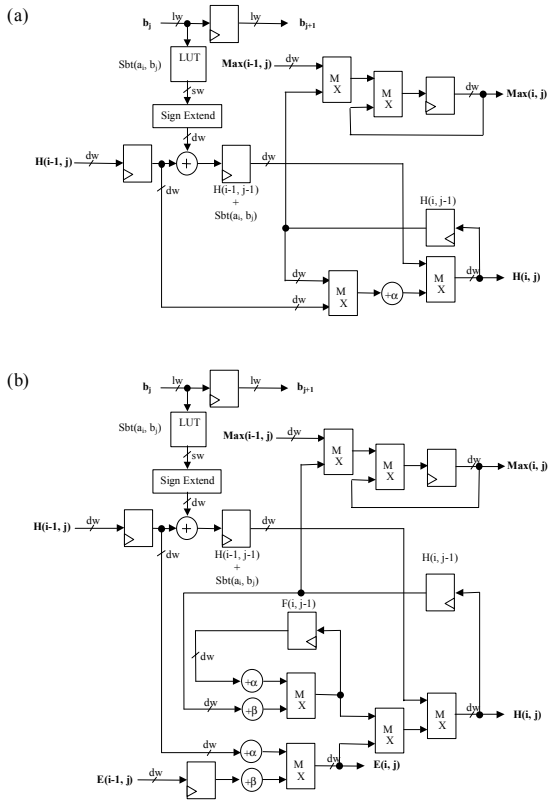

Figure 3. (a) Shows a linear gap penalty PE_i. (b) Shows an affine gap penalty PE_i.

In iteration step k, $1 \le k \le M+K-1$, the values $H(i,j)$, $E(i,j)$, and $F(i,j)$ for all i, j with $1 \le i \le M$, $1 \le j \le K$ and $k=i+j-1$ are computed in parallel in all PEs $1 \le i \le M$, within a single clock cycle. For this H is stored in PE M, which is then written into the off-chip memory.

So far we have assumed a processor array equal in size of the query sequence length. In practice, this rarely happens. Since the length of the sequences may vary (several thousands in some cases, however commonly the length is only in hundreds), the computation must be partitioned on the fixed size processor array. The query sequence is usually larger than the processor array. For sake of clarity we firstly assume a query sequence of length M and a processor array of size N where M is a multiple of N, i.e. $M=k \cdot N$ where $k \ge 1$ is an integer. A possible solution is to split the computation into k passes:

The first N characters of the query sequence are assigned to the processor array and the corresponding substitution table columns loaded. The entire database then crosses the array; the H-value and E-value computed in PE N in each iteration step are output. In the next pass the following N characters of the query sequence are loaded into the array. The data stored previously is loaded together with the corresponding subject sequences and sent again through the processor array. The process is iterated until the end of the query sequence is reached.

Unfortunately, this solution requires a large amount of memory (assuming 16-bit accuracy for intermediate results, four times the database size per pass is needed). The memory requirement

can be reduced by factor p by splitting the database into p equal-sized pieces and computing the alignment scores of all subject sequences within each piece. However, this approach also increases the loading time of substitution table columns by a factor of p. In order to eliminate this loading time we have slightly extended our PE design. Each PE now stores k columns of the substitution table instead of only one. Although this increases the area per PE (see Section 5 for details), it allows for alignment of each database sequence with the complete query sequence without additional delays. It also reduces the required memory for storing intermediate results to four the times longest database sequence size (again assuming 16-bit accuracy). Figure 4 illustrates our solution. The linear array of PEs is encapsulated in *psap_n.* The database sequences are passed in from the host one by one through a FIFO to the *S*2 interface. The database sequences have been pre-converted to LUT addresses. For query lengths longer than the PE array the intermediate results are stored in a FIFO of width $2\times dw + lw + 1$ for affine gap penalty. For linear gap penalty the FIFO width is $dw + lw + 1$. The FIFO depth is sized to hold the longest sequence in the database. The database sequence is also stored in the FIFO. On each consecutive pass an LUT offset is added to address the next column of the substitution table stored within the PEs. The maximum score on each pass is compared with those from all other passes and the absolute maximum is returned to the host.

We can again take advantage of reconfiguration and design different configurations for different values of k. This allows us to load a particular configuration that is suited for a range of query sequence lengths. calculation PE i, $2 \le i \le M$, receives the values $H(i-1,j)$, $E(i-1,j)$, and b_j from its left neighbour $i-1$, while the values $H(i-1,j-1)$, $H(i,j-1)$, $F(i,j-1)$, a_i, α, β, and $Sbt(a_i,b_j)$ are stored locally. PE 1 receives b_j in steps j with $1 \le j \le K$. Computation for linear gap penalties are similar.

Thus, it takes $M+K-1$ steps to compute the alignment score of the two sequences with the SW algorithm. However, notice that after the last character of B enters the array, the first character of a new subject sequence can be input for the next iteration step. Thus, all subject sequences of the database can be pipelined with only one-step delay between two different sequences.

Because of the very limited memory of each PE, only the highest score of matrix H is computed on the FPGA for each pair-wise comparison. The front end PC carries out ranking the compared sequences and reconstructing the alignments. Because this last operation is only performed for very few subject sequences, its computation time is negligible.

Our PE design incorporates the maximum computation of the matrix H with only a constant time penalty as follows: After each iteration step all PEs compute a new value *max* by taking the maximum of the newly computed H-value and the old value of *max* from its left neighbor. After the last character of a subject sequence has been processed in PE M, the maximum of matrix

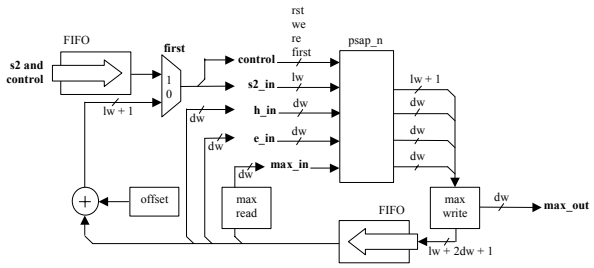

Figure 4. System Implementation

So far we have assumed that the query sequence length M is a multiple of the processor array size N, i.e. $M = k \cdot N$ where k is an integer. If this is not the case, we can still use our design by filling substitution table columns in the remaining PEs with zeros.

5. PERFORMANCE EVALUATION

We have described the PE design in Verilog and targeted it to the Xilinx Virtex II architecture. We have specified an area constraint for each PE. The linear array is placed in a zigzag pattern as shown in Figure 5. We use on-chip RAM for the partial result FIFO. The FIFO depth has been sized to 8192 entries and uses almost all of one column of block SelectRAM. Database sequences longer than 8192 are aligned on the host. The host interface takes up some of the FPGA space in the bottom right-hand corner. We allocate a 192×160 array of logic slices for the PEs. The aspect ratio of the PEs is scaled to maximize the utilization internal to the PE and minimize the fragmentation of the FPGA resource.

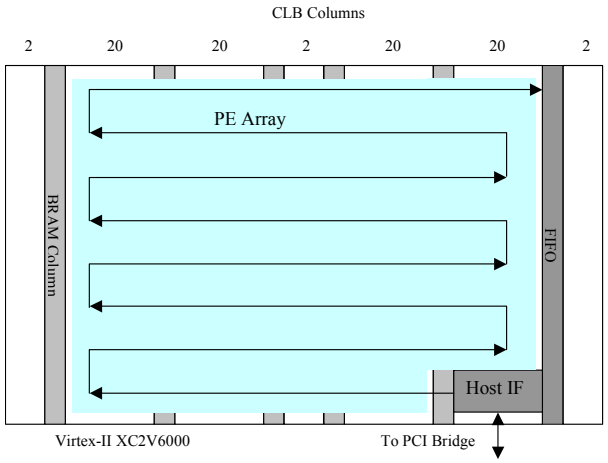

Figure 5. System Floor plan in the XC2V6000 on the Alpha-Data ADM-XRC-II Board

The size of one linear gap penalty PE is 3×10 CLBs and the size of one affine gap penalty PE is 6×8 CLBs. We have implemented a linear array of these PEs. Using a Virtex II XC2V6000 we are able to accommodate 252 linear PEs or 168 affine PEs using k=3. This allows handling of query sequence lengths up to 756 and 504 respectively, which is sufficient in most cases (74% of sequences in Swiss-Prot are ≤ 500 [3]). For longer queries we have implemented a design with k = 12, which can accommodate 168 linear PEs or 119 affine PEs. The corresponding clock frequencies are 55 MHz for linear and 44 MHz for affine.

A performance measure commonly used in computational biology is *cell updates per second* (CUPS). A CUPS represents the time for a complete computation of one entry of the matrix *H*, including all comparisons, additions and maxima computations. The CUPS performance of our implementations can be measured by multiplying the number of PEs by the clock frequency. Table 1 summarizes our results.

Since CUPS does not consider data transfer time, query length and initialization time, it is often a weak measure that does not reflect the behavior of the complete system. Therefore, we will use database scans for different query lengths in our evaluation.

Table 1. Linear array performance for different PE designs mapped to a Virtex-II XC2V6000

Design	Max PEs Fitted	Max Speed (MHz)	Max. Query length (PEs x *k*)	Peak Performance (GCUPS)
Linear, *k*=3	252	55	756	13.9
Linear, *k*=12	168	55	2016	9.2
Affine, *k*=3	168	45	504	7.6
Affine, *k*=12	119	44	1428	5.2

Table 2 reports the performance for scanning the Swiss-Prot protein databank (release 43, which contains 146'720 sequences comprising 54'093'154 amino acids [3]) for query sequences of various lengths using our design on an ADM-XRC-II FPGA Mezzanine PCI-board with a Virtex-II XC2V6000 from Alpha-Data [1]. The query sequence lengths have been chosen to illustrate the effect that length has on performance. Maximum performance is achieved when the query length is closely matched to an integer multiple of the PE-array length. Note how the performance increases as the database-streaming overhead is amortized when the PE-array is required to performs more processing passes. These scan times omit just 2 sequences in Swiss-Prot Release 43 because they are longer than 8192 amino acids.

For the same application an optimized C-program on a Pentium IV 1.6 GHz has a performance of 52 MCUPS for linear gap penalties and 40 MCUPS for affine gap penalties. Hence, our FPGA implementation achieves a speedup of approximately 170 for linear gap penalties and 125 for affine gap penalties.

Table 2. The mean performance of an affine gap penalty PE-array of length 119 (*k*=12) when scanning Swiss-Prot Release 43 for several query length ranges.

Query length range	Number of Processing Passes	Mean Performance (GCUPS)	Mean Scan Time (s)
3 – 119	1	2.6	3.8
120 – 238	2	2.8	5.2
715 – 833	7	4.8	11.4
1310 – 1428	12	5	17.9

For the comparison of different massively parallel machines, we have taken data from [7],[13],[21],[25] for a database search with the SW algorithm for different query lengths. The Virtex II XC2V6000 is around ten times faster than the much larger 16K-PE MasPar. Kestrel, Fuzion and Systola 1024 are one-board SIMD solutions. Kestrel is 12 times slower [7], Fuzion is two to three times slower [21], and Systola is around 50 times slower [21] than our solution. All these boards reach a lower performance, because they have been built with older CMOS technology (Kestrel: 0.5-µm, Fuzion: 0.25-µm, Systola 1024: 1.0-µm) than the Virtex II XC2V6000 (0.15-µm). Extrapolating to this technology both SIMD and reconfigurable FPGA platforms have approximately equal performance. However, the difference between both approaches is that FPGAs allow easy upgrading, e.g. targeting our design to a Virtex II XC2V8000 would improve the performance by around 30%.

Our implementation is slower than the FPGA implementations described in [10],[14],[26]. However, these designs only implement edit distance. This greatly simplifies the PE design and therefore achieves a higher PE density as well as a higher clock frequency. Although of theoretical interest, edit distance is not used in practice because it does not allow for different gap penalties and substitution tables. The FPGA implementation presented in [25] on a Virtex XCV2000E is around three times slower than our solution. Unfortunately, the design only implements global alignment.

6. RUN-TIME RE-CONFIGURATION

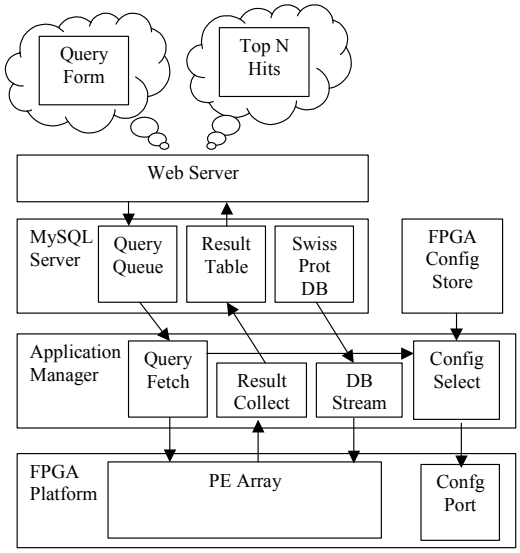

Figure 6. Web Service Architecture

We plan to make the application available as a web service. A user would enter a database query using a form-based web page similar to MPSrch [23]. The service architecture is illustrated in Figure 6. A MySQL server is used to store the Swiss-Prot database and queue user queries. The application manager handles communication between the MySQL server and the FPGA board. On start-up the sequence data is extracted from the database and compiled into a form suitable for fast streaming to the FPGA. The parameter set from a user query is given a unique ID and stored in a MySQL table. When the application manager sees the query in the list, it processes it for transfer to

the FPGA and configures the FPGA with a linear array suited to the request. The query parameters are loaded onto the linear array and the compiled database is streamed across the PCI interface. Results are collected and ranked in the application manager. When the search is finished the results are returned to the MySQL database where the web server identifies them using the unique ID and displays them to the user.

6.1 Hyper Customizing to a Query

FPGA computing performance is limited by the amount of logic resource available. If we can increase the utilization of this finite resource we can increase the performance. The queries' parameters can be used to hyper customize the systolic array to make better use of available resource. It is only worth employing run-time re-configuration if the time saving is greater than the time to re-configure. We propose that all the configurations that are necessary have been pre-compiled and the bit-streams are available on the server. Therefore we only consider the time it takes to select the correct bit-stream and configure the FPGA. It takes approximately 80ms to configure the XC2V6000 over the PCI bus. A Swiss-Prot database scan will take several seconds so it is very feasible to re-configure the FPGA for each query.

6.1.1 Constant Propagation

As the gap penalties are the same over the entire database scan, we can make the PE design flexible by having registers to store the gap penalty (2 for affine). Making the penalties constant will not only save these registers, it would allow the synthesis to optimize away some of the adder logic and free up the interconnect resource that is required to distribute these values to all the PEs.

We synthesized a k = 12 affine PE with and without fixed gap penalties. The resource saving was around 4%. For each combination of gap penalties a different configuration will have to be created and stored on the server. This is a high overhead for such a small saving.

6.1.2 Configuration Embedded Data

The content of the PE look-up tables is constant throughout the database scan. Thus, rather than loading the query into the PEs at the start of the database scan the values could be embedded in the FPGA configuration file. Since the values are stored in 64 slices used as distributed SelectRAM we can use partial re-configuration to embed them into the FPGA configuration [5]. We selected an area of 16x16 slices for the k = 12 affine PE. If the SelectRAMs are aligned so there are 16 per CLB column we will need to modify 8 configuration frames per PE column to embed the query sequence.

Each XC2V6000 frame is 984 bytes [28]. We fit 10 PE columns so the amount of data that needs to be modified is 78720 (8x10x984) bytes. In contrast it takes approximately 628µs to load the query 29988 data values for a k = 12 PE array using an in circuit interface. The interface only adds a few registers and re-uses existing circuitry. A custom application to generate the frame data would need to be created. Therefore we decided to use in-circuit loading of the look-up tables.

6.1.3 Precision Scaling

The precision of the PE array is dictated by the bit-width of the data path. The amount of resource used by the data path scales linearly with its bit width. The operations in the data path are all additions. The propagation delay of an adder scales linearly with its bit-width thus, for a lower bit-width we should be able to fit more PEs on to the FPGA area and run them faster. Figure 7 shows the result of two PE designs synthesized and mapped to the FPGA with several different bit-widths.

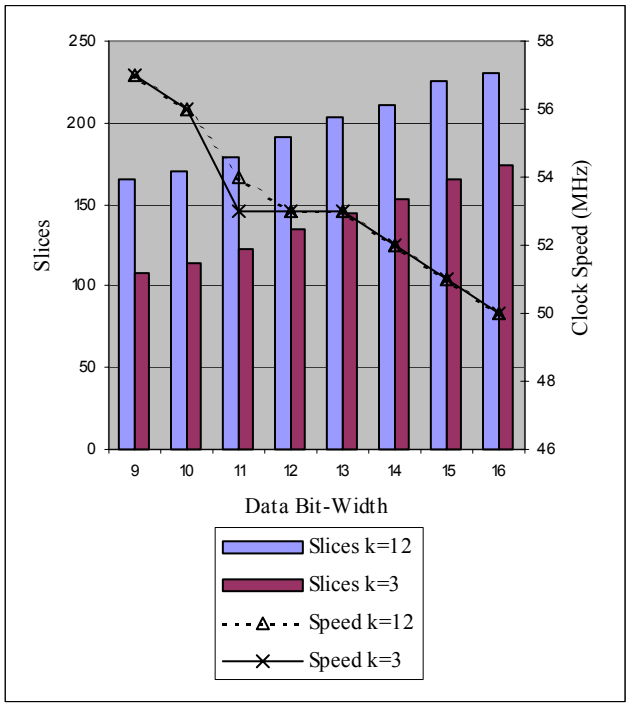

Figure 7. The Effect of Precision Scaling

Given the query parameters how do we decide on an adequate precision for the PE data path?

To answer the above question, note that the PE design uses zero limited saturation arithmetic. Therefore the minimum value is always 0 and the maximum value is $(2^N-1)/2$, where N is the bit-width. The negative values summed in the PE naturally saturate to zero. The score will only increment when two amino acids match. The largest increment in any one processing step is the highest positive value in the substitution table (Sbt_{MAX}). For example the Blosum62 matrix has a Sbt_{MAX} of 11 for matching two Tryptophan amino acids. The absolute worst case would be comparing a sequence of all Tryptophan with an equal or longer length sequence of all Tryptophan. This is clearly an unlikely occurrence but lets use it as a fast way to estimate the required precision. If M is the length of the first sequence and K is the length of the second the maximum score that could occur is $\min(M, K) \times Sbt_{MAX}$. The product of the shortest sequence length and the maximum value in the substitution table dictates the precision.

6.1.4 Dynamic Precision Scaling

Protein databases exhibit a Gaussian curve length distribution centered around 300 amino acids. The Swiss-Prot database is no exception with 50% of its sequences below 300 amino acids. The database is scanned from shortest to longest sequence. We use a lower precision PE array at the beginning, gearing up to a higher precision for longer sequences. A full device re-configuration is required each time the PE array precision is changed which takes approximately 80ms. Figure 8 shows this

dynamic scaling and how it gives us a higher MCUPS value at the start of the scan, slowing down towards the end. For a query sequence of length 1452, with the k=12 PE array, using this technique we achieved a 6% increase in the overall performance.

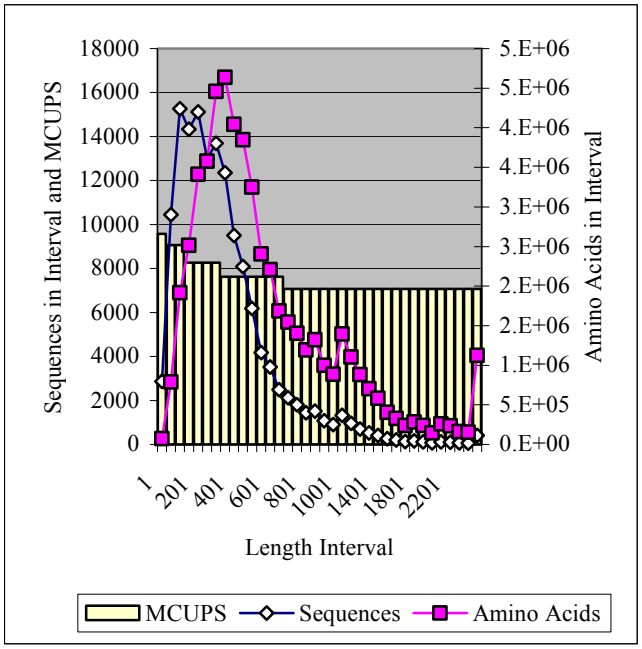

Figure 8. Dynamic precision database scan scaling to the length of Swiss-Prot database sequences

6.1.5 Partial Re-Configuration

The placement of a module in a given area of the FPGA requires that the configuration data for that module be transferred to the correct area of the configuration store. The atomicity of the configuration store strongly effects how much configuration data has to be transferred. We define an independently re-configurable region (IRR), as an area of the configuration store that can be written to effect a change in the configuration while all other regions remain running un-altered and un-disturbed. The shape of these regions is unlikely to be the same shape and size as the modules in a given system.

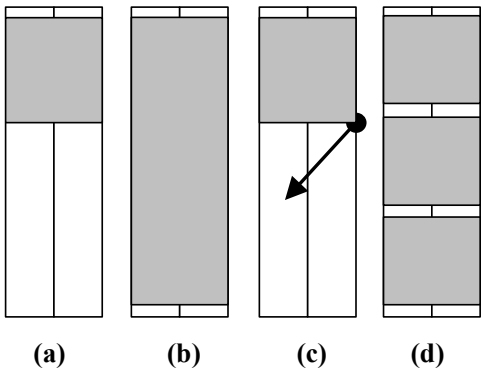

Figure 9. PEs (a) and (b) have the same configuration overhead. Shape PE to a minimum number of IRRs (c). Packing PEs into an IRR (d).

Figure 9 shows two PEs (a) and (b) that are using different amounts of FPGA resource but have the same configuration overhead due to the shape of the IRR. One solution to this issue is to try and shape a PE to better fit an IRR (c). If a long-thin aspect ratio impacts on PE performance then several are packed into an IRR (d).

The factors affecting the IRR shape of the Virtex-II architecture are the atomicity of the configuration store and the inter-module routing. The absolute atomicity of the configuration store is one frame, which spans an entire vertical slice of the device. We enforce the granularity of placement to be one CLB. Each column of CLBs requires 22 frames to be written. Communication between re-configurable modules is only possible through bus-macros [27]. The bus macro allows 4 wires per CLB row to pass across in inter-module boundary in the horizontal direction only. The bus macro also imposes the restriction that modules must be a minimum width of 4 CLBs for the Virtex-II architecture. Thus the shape of a Virtex-II IRR is 4 CLBs in the horizontal and spans the entire chip in the vertical.

Unused PEs in the array have their substitution table columns filled with zeros. Processing cycles are still used to pass the values through these unused PEs. We considered dynamically scaling the length of the PE array to the query sequence. Therefore we realigned the floor plan to fit the IRR shape of the Virtex-II FPGA as shown in Figure 10.

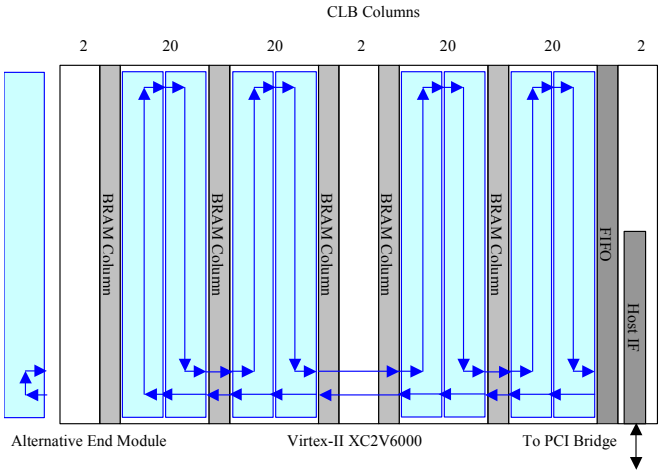

Figure 10. Floor plan in the XC2V6000 for partial re-configuration

Note the Host interface has been squeezed into the right of the device. We have created 4 unique modules that can be tiled out to create different linear array lengths. Two end-point modules are available to handle both odd and even numbers of modules. Using the 16x16 slice, k=12, 16-bit affine PE, we created a module containing 10 PEs. The Block SelectRAM limits the flexibility and amount of CLB resource we can use for the array when considering re-locatable modules. In this scenario the unused device resource would still be going to waste.

6.1.6 Multi-Threading Requests

A better use of this wasted resource would be to use it for another short PE-array. To maintain performance for short query sequences the database is split into 2 or 4 segments. We place 2 half-length or 4 quarter-length linear arrays to make better use of the FPGA resource as shown in Figure 11. The different database segments have to be streamed to each across the PCI

bus to each linear array. This could prove to be a performance bottleneck.

It is conceivable we could receive several short sequences in the request queue at any one time. To handle this we propose sorting them into bins depending on the size of PE array required. Short sequences are grouped into 2, 3 or 4 and all scheduled to the device at once. The database streaming is synchronized between the linear arrays. We can regain the 25-75% performance lost in short sequence alignments while using the same FPGA resource and database-streaming overhead.

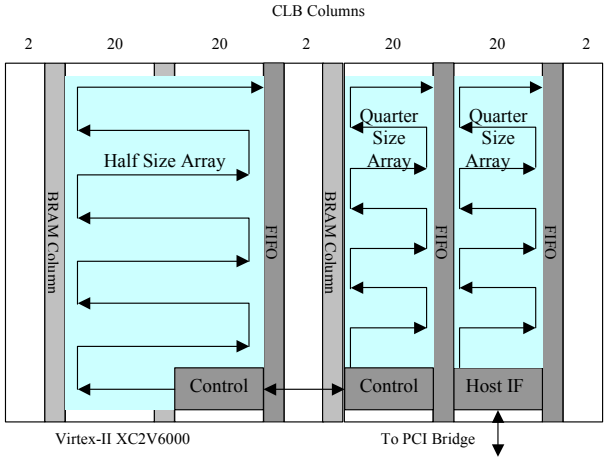

Figure 11. Half and quarter size database scanners on one Virtex-II XC2V6000

7. CONCLUSION

In this paper we have demonstrated that re-configurable hardware platforms provide a cost-effective solution to high performance bio-sequence database scanning. We have described a partitioning strategy to implement database scans with a fixed-size processor array and varying query sequence lengths. Using our PE design and our partitioning strategy we can achieve supercomputer performance at low cost on an off-the-shelf FPGA. Our FPGA implementation achieves a speedup of approximately 170 for linear gap penalties and 125 for affine gap penalties as compared to a standard desktop computing platform.We have presented several strategies for further improving the performance of our approach using hyper-customization enabled by run-time re-configuration. These strategies are constant propagation, data embedding, precision scaling, partial re-configuration and multi-threading. Our investigation suggests a 6% improvement in performance when using dynamic precision scaling during a database scan. Scheduling multiple different sized processing arrays on a single device allows us to regain the 25-75% performance lost in short sequence alignments while using the same FPGA resource and database-streaming overhead.

Our future work includes extending our design to accelerate the ClustalW multiple sequence alignment algorithm and making our implementation available as a special resource in a computational grid [19]. We will also analyze how the hyper-customization techniques presented in this paper can be used to improve the performance of other systolic array designs on FPGAs, such as the Viterbi algorithm for Hidden Markov Models.

8. REFERENCES

[1] Alpha-Data http://www.alpha-data.co.uk

[2] Altschul, S.F., Gish, W., Miller, W., Myers, E.W, Lipman, D.J.: Basic local alignment search tool, *Journal of Molecular Biology* 215 (1990) 403-410.

[3] Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M.-C., Estreicher, A., Gasteiger, E., Martin, M.J., Michoud, K., O'Donovan, C., Phan, I., Pilbout, S., Schneider, M.: The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003 *Nucleic Acids Research* 31(2003) 365-370.

[4] Borah, M., Bajwa, R.S., Hannenhalli, S., Irwin, M.J.: A SIMD solution to the sequence comparison problem on the MGAP, *in Proc. ASAP'94*, IEEE CS (1994) 144-160.

[5] Camilleri, N.: XAPP138 Status and Control Semaphore Registers Using Partial Reconfiguration, *Xilinx Inc*, Version 1.0, May 17, 1999.

[6] Chow, E., Hunkapiller, T., Peterson, J., Waterman, M.S.: Biological Information Signal Processor, *Proc. ASAP'91*, IEEE CS (1991) 144-160.

[7] Dahle, D., Grate L., Rice, E., Hughey, R.: The UCSC Kestrel general purpose parallel processor, *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Appilcations* (1999) 1243-1249.

[8] Glemet, E., Codani, J.J.: LASSAP, a Large Scale Sequence compArison Package, *CABIOS* 13 (2) (1997) 145-150.

[9] Gokhale, M. et al.: Processing in memory: The Terasys massively parallel PIM array, *Computer* 28 (4) (1995) 23-31.

[10] Guccione, S.A., Keller, E.: Gene Matching using JBits, *Proc. 12th Int. Workshop on Field-Programmable Logic and Applications* (FPL'02), Springer, LNCS 2438 (2002) 1168-1171.

[11] Guccione, S.A., Levi, D.: Run-Time Parameterizable Cores, *Proc.* 9th *Int. Workshop on Field Programmable Logic and Applications* (FPL'99), Springer, LNCS 1673, (1999) 215-222.

[12] Guccione, S.A., Levi, D., Sundararajan, P.: JBits: A Java-based Interface for Reconfigurable Computing, *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD Con'99)*

[13] Guerdoux-Jamet, P., Lavenier, D.: SAMBA: hardware accelerator for biological sequence comparison, *CABIOS* 12 (6) (1997) 609-615.

[14] Hoang, D.T.: Searching genetic databases on Splash 2, in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE CS, (1993) 185-191.

[15] Hughey, R.: Parallel Hardware for Sequence Comparison and Alignment, *CABIOS* 12 (6) (1996) 473-479.

[16] Lavenier, D., Pacherie, J.-L.: Parallel Processing for Scanning Genomic Data-Bases, *Proc. PARCO'97*, Elseiver (1998) 81-88.

[17] Lopresti, D.P.: P-NAC: A systolic array for comparing nucleic acid sequences, *Computer* 20 (7) (1987) 98-99.

[18] Pearson, W.R.: Comparison of methods for searching protein sequence databases, *Protein Science* 4 (6) (1995) 1145-1160.

[19] Project Proteus, http://www.projectproteus.com

[20] Singh, R.K. et al.: BIOSCAN: a network sharable computational resource for searching biosequence databases, *CABIOS*, 12 (3) (1996) 191-196.

[21] Schmidt, B., Schröder, H., Schimmler, M: Massively Parallel Solutions for Molecular Sequence Analysis, *Proc. 1st IEEE Int. Workshop on High Performance Computational Biology,* Ft. Lauderdale, Florida, 2002.

[22] Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences*, Journal of Molecular Biology* 147 (1981) 195-197.

[23] Sturrock, S., Collins, J.: MPsrch version 1.3, *Biocomputing Research Unit University of Edinburgh, UK*, 1993

[24] TimeLogic Corporation, http://www.timelogic.com./

[25] Yamaguchi, Y., Maruyama, T., Konagaya, A.: High Speed Homology Search with FPGAs, *Proc. Pacific Symposium on Biocomputing'02*, pp.271-282, (2002).

[26] Yu, C.W., Kwong, K.H., Lee, K.H., Leong, P.H.W.: A Smith-Waterman Systolic Cell, *Proc. 13th Int. Workshop on Field Programmable Logic and Applications (FPL'03)*, Springer, LNCS 2778, (2003) 375-384.

[27] Xilinx Inc.: XAPP290 Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations, *Xilinx Inc,* September 9, 2004.

[28] Xilinx Inc.: Virtex-II™ Platform FPGA User Guide G0U02, *Xilinx Inc,* Version 1.9, December 2002.