Multiple Sequence Alignment on an FPGA

Tim Oliver¹, Bertil Schmidt¹, Darran Nathan², Ralf Clemens², and Douglas Maskell¹

¹ School of Computer Engineering, Nanyang Technological University, Singapore 639798, tim.oliver@gmail.com, asbschmidt@ntu.edu.sg, asdouglas@ntu.edu.sg
² Project Proteus, School of Engineering, Ngee Ann Polytechnic, Singapore 599489, darran@projectproteus.org, ralf@projectproteus.org

Abstract

Molecular Biologists frequently compute multiple sequence alignments (MSAs) to identify similar regions in protein families. Progressive alignment is a widely used approach to compute MSAs. However, aligning a few hundred sequences by popular progressive alignment tools requires several hours on sequential computers. Due to the rapid growth of biological sequence databases biologists have to compute MSAs in a far shorter time. In this paper we present a new approach to MSA on reconfigurable hardware platforms to gain high performance at low cost. To derive an efficient mapping onto this type of architecture, fine-grained parallel processing elements (PEs) have been designed. Using this PE design as a building block we have constructed a linear systolic array to perform a pairwise sequence distance computation using dynamic programming. This results in an implementation with significant runtime savings on a standard off-the-shelf FPGA.

1. Introduction

Dynamic programming (DP) is often used to compute the optimal alignment of a pair of sequences [12]. However the extension of the DP method for simultaneous alignment of multiple sequences is impractical as the time and space complexities are in the order of the product of the length of the sequences. Thus, many heuristics to compute multiple sequence alignments (MSAs) in reasonable time have been developed.

Progressive alignment is a widely used heuristic [3]. Examples include ClustalW [13], PRALINE [5], and PILEUP. Typically, progressive alignment methods consist of three steps. First, a distance value between each pair of sequences is computed. Secondly, a phylogenetic tree is calculated based on this distance matrix. Finally, pairwise alignment of various profiles is done following the branching order in the phylogenetic tree to form the final MSA. Unfortunately, progressive alignment programs suffer from a high computational complexity, for instance the alignment of a few hundred protein sequences using ClustalW requires several hours on a state-of-the-art workstation.

A popular technique to speedup this time consuming task is to use parallel processing [2,7,8]. The runtime of progressive alignment programs is clearly dominated by the first step (computation of pairwise sequence distance). There are two basic approaches of parallelizing this step: one is based on the systolisation of the pairwise distance computation algorithm (fine-grained); the other is based on the distribution of the computation of pairwise distances (coarse-grained). Systolic array architectures have proven their high efficiency for pairwise sequence alignment using dynamic programming, e.g. [4,6]. Hence, they are good candidate structures for the first approach. Obviously, supercomputers and PC clusters are suitable architectures for the coarse-grained approach [2,7,8].

Special-purpose systolic arrays provide the best area/performance ratio by means of running a particular algorithm. 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 [1,10]. However, because of the high production costs involved, there are many cases where announced second-generation architectures have not been produced. The strategy in this paper is based on FPGAs. FPGAs provide a flexible platform for fine-grained parallel computing based on reconfigurable hardware. Since there is a large overall FPGA market, this approach has a relatively small price/unit and facilitates upgrading to FPGAs based on state-of-the-art technology. 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. Section 2 provides a brief description of progressive alignment using ClustalW. Section 3 describes the algorithm for pairwise sequence distance computation. The parallel algorithm and its mapping onto a reconfigurable platform are explained in Section 4. The performance is evaluated and compared to previous implementations in Section 5. Section 6 concludes the paper.
2. Progressive Sequence Alignment

In this section, we briefly describe the three stages involved in progressive alignment using ClustalW [13] as an example (see Figure 1).

Distance matrix: A distance between each pair of sequences value is computed using pairwise sequence alignment. The obtained valued from the alignments are stored in a so-called distance matrix. Section 3 explains in detail how these values are calculated.

Guided tree: This step uses the distance matrix obtained from the first step and forms a guided-tree using the neighbor-joining method [11]. The leaves of the tree contain the various sequences. The topology of the tree is totally dependent upon the sequences that are taken, i.e. closely related sequences are placed together and share a common branch in the guided-tree and divergent sequences are widely spaced in the tree. The guided-tree is used to find out closely related sequences or a group of sequences that are aligned progressively in the last step to form the final MSA.

Progressive Alignment: First closely related sequences or group of sequences are aligned and at the end most divergent sequences are aligned to get the final MSA.

Table 1: Profiling of the three stages of ClustalW using a different number of globin sequences on a Pentium4 3GHz (based on the code from [8])

<table>
<thead>
<tr>
<th>#sequences</th>
<th>Distance matrix</th>
<th>Guided tree</th>
<th>Progressive alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>87.1%</td>
<td>0.03%</td>
<td>12.8%</td>
</tr>
<tr>
<td>200</td>
<td>93.1%</td>
<td>0.07%</td>
<td>6.8%</td>
</tr>
<tr>
<td>400</td>
<td>92.1%</td>
<td>0.25%</td>
<td>7.7%</td>
</tr>
<tr>
<td>800</td>
<td>92.3%</td>
<td>0.54%</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

Profiling of the three stages of ClustalW for different numbers of globin sequences (see Table 1) reveals that more than 90% of the overall runtime is spent on the first stage (distance matrix computation). Hence, we have decided to parallelize only this stage on an FPGA.

3. Pairwise Sequence Distance Computation

Given is a set of $n$ protein sequences $S = \{S_1,...,S_n\}$. For two sequences $S_i, S_j \in S$ we define their distance $d(S_i,S_j)$ as follows:

$$d(S_i,S_j) = \frac{\text{nid}(S_i,S_j)}{\min(l_i,l_j)}$$

where $\text{nid}(S_i,S_j)$ denotes the number of exact matches in the optimal local alignment of $S_i$ and $S_j$ (with respect to the parameters $\alpha$, $\beta$ and $\text{sbt}$, which are explained below) and $l_i$ ($l_j$) denotes the length of $S_i$ ($S_j$).

The optimal local alignment of two sequences can be computed using the Smith-Waterman algorithm [12]. 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 series of such elementary operations, any segments can be transformed into any other segment.

The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments.

Consider two strings $S_1$ and $S_2$ of length $l_1$ and $l_2$. 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 $S_1$ and $S_2$. The computation of $H(i,j)$, for $1 \leq i \leq l_1$, $1 \leq j \leq l_2$, is given by the following recurrences:

$$H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1) + \text{sbt}(S_1[i],S_2[j])\}$$

$$E(i,j) = \max\{H(i,j-1) - \alpha, E(i,j-1) - \beta\}$$

$$F(i,j) = \max\{H(i-1,j) - \alpha, F(i-1,j) - \beta\}$$

where $\text{sbt}$ is a character substitution cost table. Initialization of these values are given by $H(i,0) = E(i,0) = F(0,j) = F(0,j) = 0$ for $0 \leq i \leq l_1$, $0 \leq j \leq l_2$. Multiple gap costs are taken into account as follows: $\alpha$ is the cost of the first gap; $\beta$ 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:

$$H(i,j) = \max\{0, H(i,j-1) - \alpha, H(i-1,j) - \alpha, H(i-1,j-1) + \text{sbt}(S_1[i],S_2[j])\}$$

Each position of the matrix $H$ is a similarity value. The two segments of $S_1$ and $S_2$ producing this value can be determined by a traceback procedure. Figure 2 illustrates an example.

The value $\text{nid}(S_1,S_2)$ of two sequences $S_1$ and $S_2$ can then be computed by counting the number of exact character matches during the traceback procedure of the Smith-Waterman algorithm. For instance the $\text{nid}$-value for the example given in Figure 2 is six. Unfortunately, this procedure is not very suitable for a fine-grained parallel implementation. Therefore, we have formulated a
new recurrence relation for the nid-value computation that is more suitable for implementation using a linear systolic array. It facilitates nid-calculation without computation of the actual alignment. In the following, we first explain the idea for linear gap penalties and then generalize it for affine gap penalties. Its efficient implementation using a linear systolic array architecture on an FPGA is then described in Section 4.

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

where

\[
m(i, j) = \begin{cases} 
1 & \text{if } S_i[j] = S_j[j] \\
0 & \text{otherwise} \\
\end{cases}
\]

The value \(\text{nid}(S_1, S_2)\) is then equal to \(N(i_{\max}, j_{\max})\) where \(i_{\max}, j_{\max}\) are the coordinates of the maximum in matrix \(H\). Figure 3 shows the values of the \(N(i, j)\) matrix using the sequences and parameters of the example given in Figure 2. For affine gap penalties the recurrence relation for \(N(i, j)\) is extended as follows:

\[
N(i, j) = \begin{cases} 
0 & \text{if } H(i, j) = 0 \\
N(i - 1, j - 1) + m(i, j) & \text{if } H(i, j) = H(i - 1, j - 1) + \text{sbt}(S_i[j], S_j[j]) \\
N(i, j - NE(i, j)) & \text{if } H(i, j) = E(i, j) \\
N(i - FE(i, j), j) & \text{if } H(i, j) = F(i, j) \\
\end{cases}
\]

where

\[
NE(i, j) = \begin{cases} 
1 & \text{if } E(i, j) = H(i, j - 1) - \alpha \text{ or } j = 1 \\
NE(i, j - 1) + 1 & \text{if } E(i, j) = E(i, j - 1) - \beta \\
\end{cases}
\]

and

\[
FE(i, j) = \begin{cases} 
1 & \text{if } F(i, j) = H(i - 1, j) - \alpha \text{ or } i = 1 \\
FE(i - 1, j) + 1 & \text{if } F(i, j) = F(i - 1, j) - \beta \\
\end{cases}
\]

Figure 2. Example of the Smith-Waterman algorithm to compute the local alignment between two DNA sequences ATCTCGTATGATG 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 traceback procedure delivers the corresponding alignment (shaded cells), the two subsequences TCGTATGA and TCTATCA.

Given are the two sequences \(S_1\) and \(S_2\), a linear gap penalty \(\alpha\) and a substitution table \(\text{sbt}\). The computation of \(N(i, j)\) is given by the following recurrence relation:

4. Mapping onto an FPGA platform

More than 90% of the runtime of ClustalW is dominated by distance score calculation. We parallelize this step using the systolisation of the pairwise distance computation and map it to an FPGA device. FPGA implementations of the Smith-Waterman algorithm using systolic array architectures are not new, e.g. [14]. Indeed we have taken our previous PE design [9] that performs this step, shown in Figure 4, and adapted it to MSA. The difficulty in realizing the potential performance of this solution comes from having to calculate the distance score for each of the pairwise alignments.

The distance scores are derived from the number of matching elements in the optimal pairwise alignment. The previous design only calculated the maximum score in the Smith-Waterman DP matrix. It did not directly identify the most optimal alignment suggested by this score. To do this the location of the maxima would have to be found and a trace back performed. Only then can the number of identical elements be counted. These operations are too control intensive for efficient FPGA acceleration.

However we noticed that if we add several more DP matrices to the equation we could calculate the number of identical elements using a recurrence relation. We first implemented our new algorithm in C to ascertain whether it produced equivalent results to the original ClustalW pairwise alignment stage. Once proven we then modified our PE design to perform this extra recurrence relation. The additional logic circuitry is shown in Figure 5. The two sequence characters, \(a_i\) and \(b_j\), are tested for equality. If they are equal \(N(i - 1, j - 1)\) is incremented by one. Figure 5 shows a number of selector (SEL) blocks which are used to implement the if-statements in the recurrence relation for \(N(i, j)\) given at the end of Section 3. Note that
there is a corresponding SEL block for each maximum selection (MX) block. The selection made by a SEL block depends on the selection made by the corresponding MX block. Hence the circuit can track the number of identical elements along the path that creates the maximum alignment score.

The DP calculation can be efficiently mapped to a linear array of such PEs. We assign one PE to each character of one (subject) sequence, and then shift the secondary sequence systolically through the linear chain of PEs. For the system to support subject sequences longer than the linear array the computation must be partitioned on the fixed size processor array. The subject sequence is usually longer than the processor array. For sake of clarity we assume a subject sequence of length $N$ where $N$ is a multiple of $M$, i.e. $M=k\cdot N$ where $M$ is a multiple of $N$, i.e. $M=k\cdot N$ where $k\geq 1$ is an integer. A possible solution is to split the computation into $k$ passes:

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

In our previous work [9] we took advantage of the fact that one sequence was not changed while scoring many pairwise alignments. Previously the subject sequence required $P$ cycles, where $P$ is equal to two times the product of the number of supported amino acid codes, the number of PEs and the number of passes. For example this was around 65688 cycles for each change of the subject sequence of a linear array of 119 PEs supporting 23 amino acid codes and 12 passes. For database scanning with one query sequence this was acceptable.

![Figure 4: Affine Gap Penalty PE.](image)

![Figure 5: Additions to Calculate the Number of Identical Elements.](image)

In MSA every sequence in a set has to be compared to every other sequence. If we store the full substitution table in each PE we only need to load an element select for each PE for each pass. The same example would only require 119 cycles per pass. Using this technique we have the added advantage that the subject sequence is not limited by the number of passes the linear array can support. However the increased storage requirement placed on each PE means we have to use large memory primitives in the target device to store the substitution table (Block RAMs for the Virtex-II family).

Intermediate values between passes are stored in a loop FIFO. The size of this loop FIFO limits the number of secondary sequence elements that can be processed in one computation session. The loop FIFO size is potentially further limited as now it is competing with the linear PE array for Block RAM resource. The pairwise alignment computations have to be scheduled onto the FPGA in sessions. The number of secondary sequences in a session is dictated by the size of the loop FIFO. For each session there will be $k$ passes. $N$ elements of the subject sequence are assigned to the PE array at the beginning of each pass.

5. Performance Evaluation

The PE design we have used in the performance evaluation has a 16-bit data path and 10-bit $nid$ path. The gap penalty precision has been set to 8-bit and the local PE storage allows a 32x32 element substitution table with a resolution of 16-bits. We target the Xilinx Virtex-II
architecture. Each PE requires 1 Block SelectRAM, 173 flip-flops and 436 LUTs, mapped to 274 slices. We fitted 92 PEs on the XC2V6000 device. The maximum allowable clock speed reported by the place and route step was 34MHz. This yields an estimated peak performance of 3.1 GCUPS (billion cell updates per second in the DP matrix). The number of available LUTs and registers limits the number of PEs we can fit on the device. Block SelectRAM resource is not a limiting factor.

Table 2: Runtime comparison (in seconds) of the first stage of ClustalW on a PC (Pentium4 3GHz) and our FPGA implementation (Xilinx XC2V6000) for different numbers of globin sequences.

<table>
<thead>
<tr>
<th>Number of globin sequence</th>
<th>ClustalW stage 1 (PC)</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>42.4</td>
<td>1.1</td>
</tr>
<tr>
<td>200</td>
<td>184</td>
<td>3.6</td>
</tr>
<tr>
<td>400</td>
<td>833</td>
<td>18.1</td>
</tr>
<tr>
<td>600</td>
<td>1697</td>
<td>38.2</td>
</tr>
<tr>
<td>800</td>
<td>2967</td>
<td>65.0</td>
</tr>
<tr>
<td>1000</td>
<td>4410</td>
<td>97.5</td>
</tr>
</tbody>
</table>

A set of performance evaluation tests have been conducted using different numbers of globin sequences, to evaluate the processing time of the ClustalW implementation versus that of the original ClustalW pairwise alignment stage on the PC. The PCI based ADP-WRC-II board from Alpha-Data with a Xilinx XC2V6000 FPGA has been used in the tests. This FPGA accommodates 92 PEs and is clocked at 34MHz. The ClustalW application is benchmarked on an Intel Pentium4 3GHz processor with 1GB RAM. The results for this are shown in Table 2.

6. Conclusions and Future Work

In this paper we have demonstrated that reconfigurable hardware platforms provide a cost-effective solution to multiple sequence alignment. A PE design for the computation of pairwise distances between protein sequences has been presented. We have described a partitioning strategy to implement this computation with a fixed-size linear systolic array and varying sequence lengths. The systolic array design is based on a new recurrence relation for calculating the number of exact matches in the optimal local alignment of two sequences. Our implementation can achieve a speedup of almost 50 with an off-the-shelf FPGA compared to Pentium4 3GHz for the compute-intensive first stage the popular ClustalW MSA program.

Our future work includes exploring the potential for a higher performance by pipelining the PE data path. The estimated peak performance is equal to the product of the number of PEs and the clock speed. This presents a tradeoff between the number of pipelined PEs we would be able to fit and the clock speed improvement. N elements of the subject sequence are assigned to the PE array at the beginning of each pass. The k passes have to be done in each of the S session. We could reduce this overhead by a factor of S by providing storage in every PE for the element assignment for each of the k passes. In a linear array of 92 PEs a 32 entry 5-bit store in each PE would allow a maximum subject sequence of 2944 to be supported while reducing the subject sequence loading this overhead by a factor of S.

References