Efficient Pairwise Statistical Significance Estimation using FPGAs

Daniel Honbo, Ankit Agrawal, and Alok Choudhary
Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA

Abstract - *In this paper, we present a fast pairwise statistical significance estimator using a Field Programmable Gate Array (FPGA) coprocessor. The running time of the pairwise statistical significance estimation routine is dominated by the hundreds of local alignments it must compute. By offloading the alignment task to an accelerator designed to concurrently process multiple independent alignments, we are able to increase the end-to-end performance of the algorithm by more than 200x over a baseline software implementation. Our proposed accelerator outperforms optimized, multicore software implementations and other FPGA implementations for pairwise statistical significance estimations.*

Keywords: Pairwise statistical significance, FPGA

1 Introduction

Estimating the statistical significance of a pairwise local alignment is an important problem in bioinformatics [1][2][3][4], and is a crucial step in many sequence comparison based applications in bioinformatics requiring homology detection. Statistical significance represents the likelihood that the similarity between two given sequences could have arisen by chance alone. In recent years, pairwise statistical significance [5][6][7][8] has been shown to be a promising alternative to database statistical significance (statistical significance reported by database search programs like BLAST, FASTA, etc.) for the purpose of identifying homologs. However, current implementations of the variants of pairwise statistical significance estimation are too slow to be useful in many large scale applications, such as database search, because the algorithm performs hundreds of alignments during the processing of a single sequence pair. The demonstrated significant improvement in retrieval accuracy using pairwise statistical significance strongly motivates the use of high performance computing techniques to speed-up the pairwise statistical significance estimation procedure.

In this paper, we use a large FPGA to accelerate the computationally intensive sequence alignment task performed during the pairwise statistical significance estimation procedure. An FPGA is a hardware device that can be configured at run-time to implement an arbitrary digital circuit. It excels in situations where simple operations, such as comparisons, logical operations and integer arithmetic, are performed on data streams. The regular data dependencies, abundant parallelism, and straightforward integer arithmetic of the alignment task make it suitable for implementation on an FPGA. We customize our implementation based on specific features of the pairwise statistical significance estimation procedure, which allows us to achieve significantly better performance than we would with a generic local alignment implementation.

The remainder of this paper is organized as follows. We provide an overview of the pairwise statistical significance algorithm in Section 2. Related work in the area of biological sequence alignment acceleration is summarized in Section 3. In Section 4 we present our architecture for accelerating the pairwise statistical significance algorithm. Section 5 discusses the performance results of our implementation.

2 Algorithm overview

Consider the pairwise statistical significance as described in [5] to be obtainable by the following function:

PairwiseStatSig(Seq1, Seq2, SC, N)

where *Seq1* and *Seq2* are the two sequences, *SC* is the scoring scheme (substitution matrix, gap penalties), and *N* is the number of shuffles. The *PairwiseStatSig* function first generates a score distribution by aligning *Seq1* with *N* shuffled versions of *Seq2*, and subsequently fits an Extreme Value Distribution (EVD)[1] using censored-maximum-likelihood to get the statistical parameters K and λ. Finally, it reports the pairwise statistical significance of the alignment score of *Seq1* and *Seq2* (say x) in terms of the P-value by calculating the following probability:

$$P(S \geq x) = 1 - e^{-Kmne^{-\lambda x}} \quad (1)$$

where m, n are the lengths of the sequences, and K, λ are the statistical parameters. The P-value thus represents the probability that a score greater than or equal to x could have been obtained by chance. The scoring scheme *SC* in the *PairwiseStatSig* function can be extended to use sequence-pair-specific distanced substitution matrices [6], multiple parameter sets [7], and sequence-specific/position-specific substitution matrices [8]. A comparitive summary of the advances in pairwise statistical significance can be found in [9].

[1] The distribution of Smith-Waterman alignment scores for the given scoring scheme, sequence lengths, and sequence compositions is empirically known to follow an EVD [9][4].

The bulk of the processing time for the algorithm is spent computing alignment scores, with most of the small remainder being devoted to the shuffling task (Table 1). Clearly, reducing the time required to complete the alignment task will net significant improvements to the performance of the algorithm as a whole. For this reason, we focus our attention on the alignment task.

Table 1: Breakdown of software execution time[2]

Seq. Length	Align	Shuffle	Other
128	98.6%	1.2%	0.2%
256	99.3%	0.7%	0.0%
512	99.7%	0.3%	0.0%
1024	99.8%	0.2%	0.0%

2.1 Smith-Waterman local alignment

The Smith-Waterman algorithm is a dynamic programming method for identifying the optimal local alignment between two sequences, A and B, where $A=a_1a_2...a_n$, $B=b_1b_2...b_m$, $A[1...i]=a_1a_2...a_i$ for $i<n$, and $B[1...j]=b_1b_2...b_j$ for $j<m$. The dynamic programming sub-problem is governed by the following equations:

$$G_A(i,j) = \max\begin{cases} G_A(i-1,j)+e \\ H(i-1,j)+o \end{cases} \quad (2)$$

$$G_B(i,j) = \max\begin{cases} G_B(i,j-1)+e \\ H(i,j-1)+o \end{cases} \quad (3)$$

$$H(i,j) = \max\begin{cases} H(i-1,j-1)+d(a_i,b_j) \\ G_a(i,j) \\ G_b(i,j) \\ 0 \end{cases} \quad (4)$$

The algorithm is based on the observation that the optimal alignment of A and B can be determined from the optimal alignment of substrings of A and B. Specifically, the optimal alignment $H(n,m)$ is the maximum of four alternatives: (1) the optimal alignment of $A[1...n-1]$ and $B[1...m-1]$, plus the similarity between a_n and b_m; (2) the optimal alignment of $A[1...n-1]$ and $B[1...m]$, plus the cost of adding a gap on A; (3) the optimal alignment of $A[1...n]$ and $B[1...m-1]$, plus the cost of adding a gap on B; or (4) 0.

In the affine gap model [12], opening a new gap is penalized much more heavily than extending an existing gap. This favors alignments having fewer long gaps over those having many short gaps. Typically, the score associated with opening a gap, o, is set somewhere around -10, while the score for extending a gap, e, is set to about -2. The implication of two separate gap penalties is that the high cost of opening a gap is amortized over the length of the gap. As such, it is generally not known whether the gap will be worth creating until it has been extended to some length. The dynamic programming algorithm must therefore keep track of 3 alignments at all times: the current optimal alignment, H, the optimal alignment constrained to end with a gap on A, G_A, and the optimal alignment constrained to end with a gap on B, G_B.

The Smith-Waterman algorithm outputs the optimal local alignment, which is the maximum value of matrix H. The associated optimal alignment, if required, can be recovered by tracing back through H, starting at the location of the maximum value, until a value of 0 is encountered. The complexity of the algorithm is $O(mn)$.

3 Related work

There are numerous examples of FPGA-based accelerators for local alignment, including those presented in [13][14][15][16]. From the operational standpoint, they differ based on the type of sequence comparison performed (DNA or protein), the supported gap model (linear or affine), and the supported sequence length. Protein alignments demand more resources than DNA alignments, and supporting an affine gap model is more complicated than a constant or linear model. The maximum supported sequence length also has a significant effect on the resource requirements of an implementation. Our implementation supports long protein alignments with an affine gap model, making it very resource-intensive among FPGA implementations.

Systolic arrays are overwhelmingly the processing paradigm of choice for FPGA implementations of sequence alignment algorithms. A systolic array is an interconnected network of processing elements, or cells, in which each cell takes data in from one or more of its neighbors, performs some computation, and passes the results along to other neighbors. Systolic arrays map well to FPGAs and are useful in situations where data parallelism is abundant and data dependencies are regular.

Typically, linear arrays are used to process a single alignment at a time. Each cell holds on to a specific element of A, while B streams through the array. While the theoretical peak performance of linear systolic arrays is often impressive, performance tends to be poor for short sequences due to low utilization of processing elements. This, in addition to data transfer overhead, is the reason why FPGA implementations commonly report a large discrepancy between the actual performance for short sequences and the theoretical performance of the array as a whole. In [15], various methods involving runtime reconfiguration are proposed to adapt the array to the length of the input query, in order to address the problem of low utilization for long arrays. Our implementation, by contrast, takes advantage of the task parallelism inherent in the pairwise statistical significance estimation procedure and uses straightforward architectural methods to minimize

[2] For number of shuffles, N, equal to 1000 [5][6][7][8].

transfer overhead and improve array utilization with a single FPGA configuration.

Recent advances in the programmability of Graphics Processing Units (GPUs) have also given rise to high-performance local alignment implementations [17][18]. GPU implementations map individual alignment tasks to lightweight GPU threads or thread blocks, and process a large number of tasks concurrently. The GPU typically needs thousands of active threads in order to distribute the computation across its many processing elements and hide memory access latency. The implementation in [17], for example, indicates that 28,800 threads are optimal for performance. These implementations thus perform well for applications like database searches, where a very large number of alignments are performed for a single query. Pairwise statistical significance estimations, by contrast, require a comparatively small number of alignments.

A few highly optimized software alignment algorithms using Streaming Single Instruction Multiple Data Extensions (SSE) have also been proposed. SSE is a set of extensions to the x86 architecture that enable the processor to perform an operation on multiple data elements simultaneously. Using SSE extensions can yield sizeable performance benefits for software routines exhibiting abundant data parallelism. SSE implementations of local alignment algorithms, such as the ones described in [19][20][21], leverage SSE to simultaneously update multiple cells in the similarity matrix.

The performance gains of SSE implementations are due in large part to a compact representation of scores. Since the value of any element in *H* stays smaller than 255 throughout many alignment tasks, representing scores as bytes usually poses no problem. When an overflow is detected for an alignment using byte representations, the query is re-executed with 2-byte representations of scores. In this way, the processor is able to simultaneously perform either 16 operations when scores are represented as bytes, or 8 operations when scores are represented as 2 bytes. The efficiency gains from smaller score representations far offset the penalty of reprocessing a small number of the alignments. Further performance benefits can be achieved by splitting the alignment tasks among multiple cores or multiple processors.

4 Implementation

The alignment task for pairwise statistical significance estimations has a very important feature: a fairly large number of shuffles, or random permutations, of one sequence, *Seq2*, are aligned to a common sequence, *Seq1*. Because we are interested in the total time required to compute all of these alignments, we are presented with two opportunities for acceleration: computing each alignment faster by exploiting data parallelism, or processing multiple alignments simultaneously, taking advantage of task parallelism.

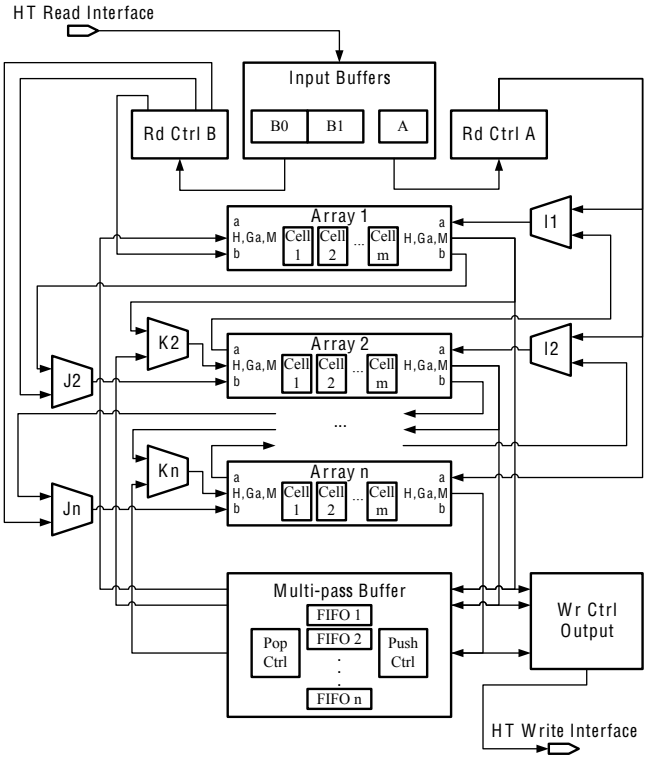

Figure 1: Accelerator Top View

The lengths of *Seq1* and *Seq2* are typically in the neighborhood of a few hundred characters, and for sequences this short, long systolic arrays tend to exhibit poor performance due to low utilization. To address this problem, we propose a flexible accelerator, shown in Figure 1, capable of behaving as a long array for long sequences or multiple shorter arrays for shorter sequences. This flexibility increases the performance of the accelerator for short sequences without sacrificing performance for the long sequences that may occasionally need to be processed. By providing architectural support for dynamic allocation of systolic cells, our flexible accelerator provides excellent actual throughput for a wide range of sequence lengths with a single FPGA bitstream configuration.

4.1 Pipelined systolic cell

A linear systolic array, which is the foundation of our implementation, accelerates the computation of a single alignment by exploiting data parallelism. We begin with a basic systolic cell similar to those described in [15][16]. Each cell is capable of calculating *H(i,j)* for a fixed *i* at every main clock cycle.

Pipelining the systolic cell, as shown in Figure 2, improves operating frequency by breaking up its critical path, and also provides the potential for an increase in throughput. The resulting data hazard on *H,* introduced because it now takes two cycles for the value of *H* to be updated, can be avoided by concurrently processing pairs of permutations, interleaving them as they are passed into the

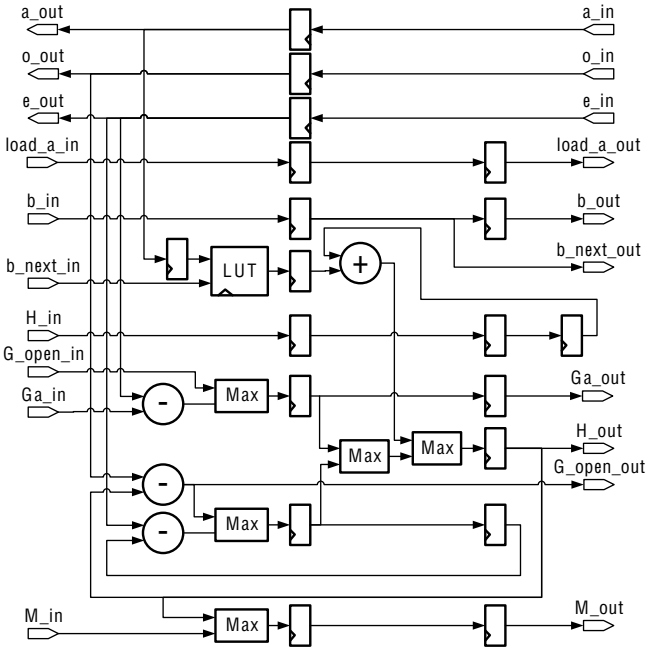

Figure 2: Pipelined Systolic Cell

cells. In this situation, we are using the pipelined systolic cell to take advantage of task parallelism.

Connecting a series of systolic cells back-to-back creates a basic linear systolic array. To process an alignment with such an array, the common sequence is shifted into the tail end of the array through cell input *a_in*. A pair of permutations is then streamed through the head of the array through cell input *b_in*. The maximum alignment score seen by each cell shifts out of the tail end of the array through the output *M_out*. The score of the alignment is the maximum value seen on the *M_out* port at the tail end of the array.

The similarity matrix *d* is implemented by the LUT component in Figure 2. The LUT is a block RAM component initialized with the values of the desired matrix. We use the general compression scheme presented in [16] to reduce block RAM utilization. For our target device family, the inputs to the block RAM are always registered. We also register the output to balance the critical path of the cell.

4.2 Flexible array support

Instead of creating a single very long systolic array, our implementation creates *n* smaller arrays, or processing blocks, where *n* is a power of 2. At run-time, the processing blocks can be configured as *n* independent arrays, as a single large array, or some configuration in between the two extremes. The configuration is accomplished by multiplexing the inputs to the head and tail cells in each processing block. The select signals of these multiplexers are tied to a software-accessible register and are written before the input data is dispatched to the accelerator.

When the processing blocks are configured as independent arrays, multiple pairs of permutations, which we collectively refer to as permutation blocks, are processed concurrently. By allowing multiple array configurations, we are providing an opportunity to trade task parallelism for data parallelism. Instead of permanently devoting all of the systolic cells to the scoring of a single alignment, we provide the option of distributing the available resources to concurrently score multiple alignments. This is an important feature for shorter sequences, where using a single long array often results in low utilization due to the natural filling and draining of the systolic array, and also to mismatches between the length of the common sequence and the number of systolic cells in each array.

To support the dynamic array configuration, the multi-pass buffer, which holds intermediate data between passes over *Seq2*, the read controllers, which direct the input sequences into the array, and the output write controller, which collects the output alignment scores, adapt their behavior based on the current array configuration.

4.3 Processing overview

Now that we have described the major components of the accelerator, we can step through the operation of the accelerator as a whole. First, the pairwise statistical significance estimation routine running on the CPU configures the accelerator by writing the length of the common sequence, the length of each permutation, the penalty for opening a gap, the penalty for extending a gap, the desired configuration of processing blocks, and the number of permutations. The software transfers the contents of the common sequence into buffer A of the accelerator, followed by the first permutation block into buffer B0 of the accelerator.

When the transfer of the first permutation block has completed, the software directs a transfer of the next permutation block into B1 while the accelerator processes the contents of B0. The software and the accelerator trade buffers when they are both done with their respective tasks, and perform their tasks again on the new buffers. This process continues until all permutation blocks have been dispatched the FPGA.

At this point, the software waits until the output write controller signals the end of processing, and transfers the output alignment scores from the output buffer of the accelerator to the CPU's memory space.

5 Results

Our test system is an XtremeData XD1000 Development system [22]. It is a Sun ULTRA 40 workstation in which one of the two processor sockets is occupied by an Altera Stratix II EP2S180 FPGA [23]. The other socket in the system holds an AMD Opteron 248

processor, which controls 4 GB of the total 8 GB DDR SDRAM on the system. The OS on the development system is Fedora Linux.

The FPGA implementation was compiled using the Quartus II software suite, version 9.1. 256 total systolic cells fit on the device, split into 8 processing blocks. The design uses 92,528 LUTs, 77,288 registers, and 8.0 million BRAM bits. It operates at 125 MHz and supports sequence lengths up to 65,535. The processing blocks can be configured as 1, 2, 4, or 8 independent arrays. The selection of 8 as the maximum number of independent arrays was made to provide a variety of potential configurations without adversely affecting cell count and performance on the target device. The substitution matrices implement the BLOSUM62 matrix.

A common metric of performance in this problem domain is a Cell Update Per Second (CUPS). A cell update refers to the process of updating *H(i,j)* for a particular *i* and *j*. Since our implementation is comprised of 256 cells, each of which is capable of performing one cell update per clock cycle, its peak performance is 32 billion CUPS (GCUPS).

To verify the functional correctness of the accelerator, we compared the scores generated by the accelerator to the scores generated by the software-only alignment task using the same common sequence and sequence of permutation blocks. Since the Smith-Waterman algorithm has a unique solution for given inputs, it is guaranteed that the quality of results for the pairwise statistical significance estimation will be unaffected by the use of this accelerator.

5.1 Accelerator performance

Figure 3 plots the comparative performance of our flexible accelerator. The numbers represent performance in GCUPS for the alignment of a common sequence *Seq1* against 1000 permutations of an equal-length sequence *Seq2*. The task of permuting *Seq2* is excluded from this analysis.

The line labeled “FPGA Flexible Array” plots the performance of our flexible accelerator. The processing blocks are configured as 8 arrays for sequences of length less than 4096, 4 arrays for sequences of length less than 8192, 2 arrays for sequences of length less than 16384, and 1 array otherwise. In other words, we use maximum number of arrays possible for the input length.

The line labeled “Software Baseline” plots the performance of a sequential, scalar implementation of the Smith-Waterman algorithm for protein sequences with support for affine gaps. The runs were conducted on the XD1000 development system. This implementation achieves at most 155 million CUPS (MCUPS).

The line labeled “Software FastFlow” measures the performance of the implementation presented in [21] on a test system equipped with an Intel Core 2 Quad Q6600 processor. To our knowledge, this is the fastest software

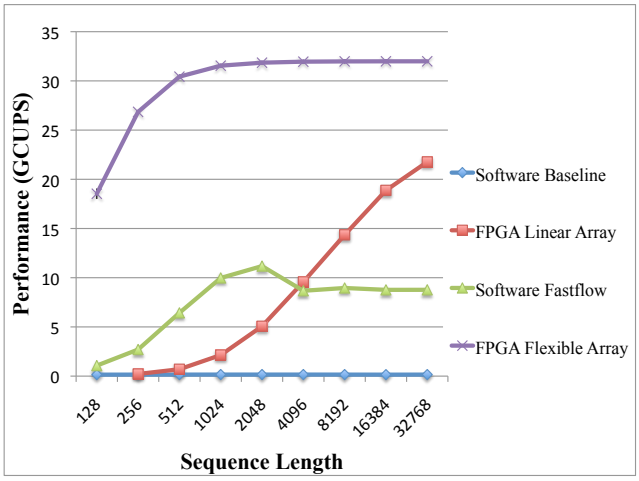

Figure 3: Accelerator Performance for 1000 Alignments

implementation available as of this writing, and is even capable of outperforming a GPU-based implementation [18] for full database searches.

Direct comparisons between FPGA implementations are difficult for a few reasons. First off, targeting DNA sequences instead of protein sequences results in a simpler cell design with much lower BRAM requirements. The choice of gap model also has a substantial effect on the complexity of the systolic cell design, with the affine gap model requiring a more complicated cell than a constant or linear gap model. Comparing GCUPS ratings from implementations such as the ones in [13] and [14], which operate on DNA and implement constant gap support, are therefore not telling.

Additionally, reducing the maximum supported score decreases the resource utilization of the systolic cell. This is because the width of the data values being operated on determines the resources needed to implement each addition and max operation, as well as the cell's register count. Since the systolic cells collectively account for the vast majority of resources used by the accelerator, decreasing their resource utilization allows for more cells on the device and higher peak performance. On top of all this, the resources available on the target FPGA, as well as the operating frequency allowed by the FPGA, are huge determining factors of the accelerator's performance.

Qualitatively, compared to the linear array presented in [15], our flexible array supports longer query sequences and achieves better peak performance, albeit with a larger FPGA. The supported query (common sequence) length for our implementation is also not constrained by the number of systolic cells, so our implementation does not require runtime reconfiguration for varying query lengths.

The implementation presented in [16] provides a good quantitative comparison to our flexible array. Like our accelerator, it handles protein sequences and supports the affine gap model. It also handles similar sequence lengths (65536). The cell design is highly optimized, and, most

importantly, it targets the same FPGA and host platform that we do, allowing an evaluation of the design and not the FPGA capacity and platform overhead. The line labeled "FPGA linear" plots the performance of this implementation based on reported results. These extrapolated numbers assume that computing *N* alignments would take *N* times as long as computing a single alignment. It should be noted that this is most likely a pessimistic assumption, since configuration registers and the common sequence should only need to be transferred to the accelerator for the first alignment. As a result, the performance for very short queries may be underestimated to some extent.

The performance results indicate that our flexible array offers significant performance advantages over software and single linear systolic arrays. The single linear array fails to outperform the FastFlow implementation for sequences under length 4096 in this case, while our flexible array is significantly faster over the entire input space, and achieves 84% of its peak throughput for sequences as short as 256 residues.

For our flexible array, the primary performance limiter for short sequences is the large contribution of data transfer overhead. The data cannot be supplied to the FPGA fast enough when sequences are very short. Also, the overhead associated with configuring the accelerator, transferring inputs and results, and managing the state of the accelerator amount to a higher percentage of total execution time when the sequences are short.

Increasing the number of permutations, *N*, beyond 1000, will have no negative effect on the average throughput of our flexible array. The time required to dispatch and compute 2*N* permutations is twice the time required to compute *N* permutations. But the costs of initializing the FPGA for computation, dispatching the first permutation block to the FPGA, and reading the results back from the FPGA, are amortized over twice the computation time. Decreasing *N*, inversely, results in overhead being amortized over fewer cycles and causes a drop in average throughput.

5.2 Application speedup

Table 2 displays the end-to-end speedup for the pairwise statistical significance estimation algorithm over a sequential, scalar software implementation. For these tests, the processing blocks were again set as multiple independent arrays.

In general, the results indicate substantial speedups. Shorter sequences see lower speedups because the speedup for the alignment task is limited by data transfer overhead, and because the unaccelerated shuffling task accounts for a higher percentage of the overall execution time.

Table 2: End-to-end Speedup

Sequence Length	Software Time (s)	Accelerated Time (s)	Speedup
128	.113	.00456	24.8
256	.441	.00859	51.4
512	1.74	.0167	104
1024	6.12	.0344	201
2048	27.6	.136	204
4096	110	.538	205
8192	450	2.15	209
16384	1,860	8.59	216

6 Conclusions and future work

In this paper, we have described an efficient pairwise statistical significance estimator using an FPGA coprocessor. Our flexible systolic array is capable of selectively trading off task-level parallelism for data parallelism, providing high throughputs for short sequences without sacrificing performance for long sequences. The configuration of the array can be adjusted at run-time by simply writing a register value to the FPGA, and does not require the FPGA to be reconfigured at runtime. We have demonstrated measured performance as high as 32 GCUPS for the accelerator, and have shown resulting end-to-end speedups over 200x.

This implementation can be readily applied to small database searches, for which pairwise statistical significance has been shown to give significantly better results than popular database search programs like BLAST, PSI-BLAST, and SSEARCH. Alternatively, it can be used to refine the results returned by these tools.

Also, the general task of query database searches, as is done with tools like FASTA and SSEARCH, is similar in nature to the alignment task of the pairwise statistical significance estimation procedure, in that both require the alignment of a common sequence against many other sequences. The parallelization methods implemented here can be applied toward database searches by concurrently processing database sequences of similar lengths against the common query sequence. Doing so should provide similar performance benefits for short sequences.

Our current FPGA implementation is designed to work for standard Smith-Waterman local alignment with affine gap penalties, and standard substitution matrices. Future work includes extending the design to incorporate more biologically relevant features in the pairwise statistical significance estimation procedure, such as the use of multiple parameter sets, and sequence- and position-specific substitution matrices.

We are also investigating methods for improving our accelerator. For example, the arithmetic units in the processing elements can be replaced by connected pairs of arithmetic units in order to handle either two independent

operations with a small range of values, or a single operation at a larger range. This would have a minimal impact on resource utilization because it only requires a few multiplexers. Much like existing SSE implementations, all permutations would be run through the coprocessor at the lower range, and any failing alignments would be rerun at full range. Since the scores generated during the alignment task of the pairwise statistical significance estimation routine are typically very small, very few alignments would need to be reprocessed.

Finally, we are working on moving the shuffling task onto the FPGA, in order to reduce transfer overhead and accelerate the shuffling process itself. This should improve the end-to-end speedup of the pairwise statistical significance estimation routine.

7 References

[1] Waterman, M.S., Vingron, M.: Rapid and Accurate Estimates of Statistical Significance for Sequence Database Searches. PNAS, USA 91(11) (1994) 4625-4628.

[2] Altschul, S.F., Gish, W.: Local Alignment Statistics. Methods in Enzymology 266 (1996) 460-80

[3] Pearson, W.R.: Empirical Statistical Estimates for Sequence Similarity Searches. JMB 276 (1998) 71-84

[4] Mott, R.: Accurate Formula for P-values of Gapped Local Sequence and Profile Alignments. JMB 300 (2000) 649-659

[5] Agrawal, A., Brendel, V.P., Huang, X.: Pairwise Statistical Significance and Empirical Determination of Effective Gap Opening Penalties for Protein Local Sequence Alignment. IJCBDD 1(4) (2008) 347-367

[6] Agrawal, A., Huang, X.: Pairwise statistical significance of local sequence alignment using substitution matrices with sequence-pair-specific distance. In: Proc. ICIT. (2008) 94-99

[7] Agrawal, A., Huang, X.: Pairwise Statistical Significance of Local Sequence Alignment Using Multiple Parameter Sets and Empirical Justification of Parameter Set Change Penalty. BMC Bioinformatics 10(Suppl 3) (2009) S1

[8] Agrawal, A., Huang, X.: Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices. IEEE/ACM TCBB (2009) 25 Sept. 2009.

[9] Agrawal, A., Choudhary, A., Huang, X. 2010. Sequence-Specific Sequence Comparison Using Pairwise Statistical Significance. In *Software Tools and Algorithms for Biological Systems,* Springer (in book series, Advances in Experimental Medicine and Biology, AEMB), 2010, in press).

[10] Karlin, S., Altschul, S.F.: Methods for Assessing the Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes. Proceedings of the National Academy of Sciences, USA 87(6) (1990) 2264-2268

[11] Smith, T.F., Waterman, M.S. 1981. Identification of common molecular subsequences. Journal of Molecular Biology 147, 195-197.

[12] Gotoh, O. 1982. An improved algorithm for matching biological sequences. Journal of Molecular Biology 162, 705-708.

[13] Hoang, D.T. 1993. Searching genetic databases on Splash 2. In IEEE Workshop on FPGAs for Custom Computing Machines, 185-191.

[14] Yu, C.W., Kwong, K.H., Lee, K.H., Leong, P.W.H. 2003. A Smith-Waterman systolic cell. In Proc. 13th Int. Workshop on Field Programmable Logic and Applications, 375-384.

[15] Oliver, T.F., Maskell, D.L. 2006. Reconfigurable architectures for bio-sequence database scanning on FPGAs. IEEE Trans. Circuit Syst. II, 52:851:855.

[16] Zhang, P., Tan, G., Gao, G. R. 2007. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In Proc. 1st intl HPRCTA '07. ACM, New York, NY, 39-48.

[17] Manavski, S, Valle, G. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, Vol 9, Suppl 2, S10.

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

[19] Farrar, M. 2007. Striped Smith--Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23, 2 (Jan. 2007), 156-161.

[20] Szalkowski A, Ledergerber C, Krähenbühl P, Dessimoz C. 2008. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. BMC Res. Notes, 1:107.

[21] Aldinucci, M, Meneghin, M, Torquati, M. 2009. Efficient Smith-Waterman on multi-core with FastFlow. To appear in: Proc. of Intl. Euromicro PDP 2010.

[22] XtremeData Inc. XD1000 development system. http://old.xtremedatainc.com/index.php?option=com_content&view=article&id=109&Itemid=170.

[23] Altera Corp. Stratix II device family. http://www.altera.com/products/devices/stratix-fpgas/stratix-ii/stratix-ii-index.jsp.