A BANDED SMITH-WATERMAN FPGA ACCELERATOR FOR MERCURY BLASTP

Brandon Harris[†], Arpith C. Jacob*, Joseph M. Lancaster*, Jeremy Buhler*, Roger D. Chamberlain*[†]

*Dept. of Computer Science and Engineering, Washington University in St. Louis
[†]BECS Technology, Inc., St. Louis, Missouri
email: {bbh2,jarpith,jmlancas,jbuhler,roger}@cse.wustl.edu

ABSTRACT

Large-scale protein sequence comparison is an important but compute-intensive task in molecular biology. The popular BLASTP software for this task has become a bottleneck for proteomic database search. One third of this software's time is spent executing the Smith-Waterman dynamic programming algorithm. This work describes a novel FPGA design for *banded Smith-Waterman*, an algorithmic variant tuned to the needs of BLASTP. This design has been implemented in Mercury BLASTP, our FPGA-accelerated version of the BLASTP algorithm. We show that Mercury BLASTP runs 6-16 times faster than software BLASTP on a modern CPU while delivering 99% identical results.

1. INTRODUCTION

Comparison of protein sequences is an important tool in modern molecular biology. Comparing newly discovered proteins to each other, or to databases of known proteins, can help to identify their functions. However, the computational cost of such comparisons has become problematic, as sequence databases such as NCBI's GenBank has grown exponentially over the last two decades.

One way to reduce the cost of proteomic sequence comparison is to accelerate it using specialized hardware. The traditional target for acceleration is the Smith-Waterman (S-W) algorithm [1]. Previous S-W implementations based on FPGAs [2, 3] have achieved speedups of 10- to 100-fold over a general-purpose CPU.

The popular BLASTP software [4] uses a *seeded alignment* heuristic to limit S-W comparison to pairs of proteins that are *a priori* likely to be highly similar. This heuristic avoids almost all the work that S-W would normally perform while still delivering results of sufficient quality to satisfy biologists. BLASTP has therefore become the dominant tool for proteomic comparison in the biological community.

In this work, we have implemented a S-W accelerator as part of Mercury BLASTP, a combined hardware/software architecture for BLASTP. There are two key distinctions between S-W as used in BLASTP and the classical algorithm. Firstly, whereas classical accelerators compare a query protein sequence to a continuous stream of other sequences, BLASTP instead delivers a stream of comparison tasks, each with its own pair of sequences to compare. Secondly, whereas classical S-W compares two entire proteins, BLASTP tries to minimize the region within each protein inspected by the algorithm. We accommodate both differences by implementing a *banded* version of S-W, which strictly limits the region explored by the algorithm to reduce computational cost. Our implementation of can scan a database of protein sequences at a rate of 282 million residues per second.

A limitation of previously published accelerators for proteomic comparison is that few have included measurements of their *sensitivity* relative to the BLASTP software. Without such measurements, there is little reason for biologists to trust the results produced by these systems. We therefore demonstrate that our complete BLASTP implementation can compete on *both* speed and sensitivity. In particular, we achieve throughput 6-16 times faster than the BLASTP software on a modern CPU while delivering results 99% identical to those returned by the software.

1.1. Related work

Acceleration of the BLAST family of algorithms requires acceleration of several computations, not just S-W. Mercury BLASTP's FPGA architectures for those other computations are described in [5, 6]. Mercury BLASTN, our accelerator for BLAST on DNA sequences, does not include a hardware S-W implementation.

In addition to the S-W accelerators described above, recent literature reports several accelerators for BLAST-like seeded alignment algorithms [7, 8]. None of these accelerators include a S-W stage, though [8] plans one for future work[1]. Moreover, none of them provide sensitivity measurements for their implementations versus the software that they are designed to replace.

This research was supported by NIH/NGHRI grant 1 R42 HG003225-01 and NSF grants CCF-0427794 and DBI-0237902. R.D. Chamberlain is a principal in BECS Technology, Inc.

[1] In some cases, the accelerator targets only DNA comparison, for which Smith-Waterman is not a bottleneck.

2. BACKGROUND: SMITH-WATERMAN AND ITS ROLE IN BLASTP

Proteomic comparison algorithms compare a *query* protein to a *subject* protein, which is drawn from a database of subjects. For our purposes, proteins are strings of *residues*, or characters from the 20-character amino acid alphabet.

2.1. Classical Smith-Waterman

S-W computes an *alignment* between query and subject, which matches up pairs of residues in the two sequences that likely evolved from a common ancestral residue. Each pair of aligned residues x, y is assigned a score $\delta(x, y)$, with more biologically similar pairs receiving higher scores. Runs of $k \geq 1$ residues in one protein with no corresponding residues in the other protein are *gaps*, which receive a penalty $-g_o - k \cdot g_e$. S-W finds an alignment with the best total score among all possible alignments of query and subject. The alignment found is *local*; that is, it may align only a substring of each protein. Higher-scoring alignments provide stronger evidence that the two aligned proteins are biologically related.

Formally, let x and y be sequences of lengths m and n, and let $M_{i,j}$ be the score of an optimal local alignment between substrings $x[1..i]$ and $y[1..j]$ ($0 < i \leq m$, $0 < j \leq n$). We may compute $M_{i,j}$ by the following dynamic programming recurrence:

$$\begin{aligned} M_{i,j} &= \max\{M_{i-1,j-1} + \delta(x[i], y[j]), I_{i,j}, D_{i,j}, 0\} \\ I_{i,j} &= \max\{M_{i,j-1} - g_o, I_{i,j-1}\} - g_e \\ D_{i,j} &= \max\{M_{i-1,j} - g_o, D_{i-1,j}\} - g_e. \end{aligned}$$

$M_{i,j}$ is computed as the best of four possibilities: the best alignment may align residues $x[i]$ and $y[j]$, or it may leave either $x[i]$ or $y[j]$ unaligned, or (if all these possibilities yield alignments with negative scores) it may leave the sequences unaligned with score 0. The I and D portions of the recurrence track the scores of optimal alignments ending with a gap in x and y, respectively. The recurrence is initialized with $M_{0,j} = M_{i,0} = 0$ and $I_{0,j} = D_{i,0} = -\infty$. An optimal local alignment then has score $\max_{i,j} M_{i,j}$, which is computed in time $\Theta(mn)$.

Let the set of three values $M_{i,j}$, $I_{i,j}$, and $D_{i,j}$ be the i, j*th cell* of the computation. Cell i, j is dependent on only three other cells for its value, namely cells $i-1, j$; $i, j-1$; and $i-1, j-1$. Hence, all the cells with a common value of $i+j$ may be computed in parallel without violating any data dependencies of the recurrence. We call this set of cells the $i+j$*th anti-diagonal* of the recurrence. Traditional hardware accelerators for S-W are organized as a systolic array that computes successive anti-diagonals of the recurrence in a pipelined fashion.

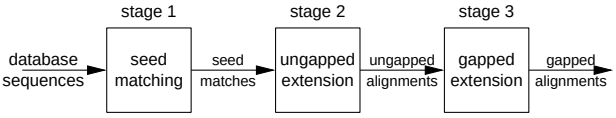


Fig. 1. The BLASTP computational pipeline.

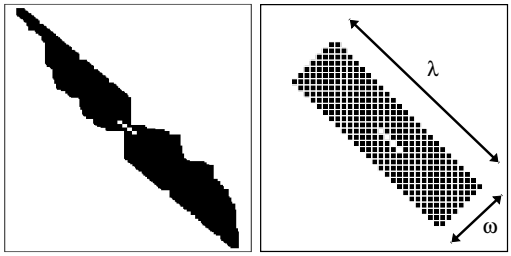
2.2. The BLASTP Pipeline

As shown in Figure 1, BLASTP consists of a pipeline with three stages: *seed matching*, *ungapped extension*, and *gapped extension*. The seed matching stage identifies short patterns of residues, called *seed matches*, that appear in both query and subject. Each seed match is described by its coordinates i, j in query and subject, respectively. Seed matches are forwarded to ungapped extension, which aligns the query and subject with the restrictions that the alignment must intersect the seed match positions i and j and must not contain gaps. Any sufficiently high-scoring ungapped alignment from this step is called a *high-scoring segment pair*, or HSP. HSPs are passed on to gapped extension, where the two proteins are (at last) aligned with S-W, which does allow for gaps. The execution profile of the BLASTP software [4] shows that about a third of CPU time is still spent in gapped extension. Hence, meaningful acceleration of BLASTP still requires acceleration of S-W.

2.3. Smith-Waterman within BLASTP

While classical S-W does not restrict its search space, gapped extension restricts the alignments sought by S-W to those that pass through or near a seed match. NCBI BLASTP computes two optimal alignments: one alignment *ending* at the match positions i, j (and starting anywhere before them), and another *starting* at i, j (and ending anywhere after them). Concatenating these two alignments yields an optimal alignment constrained to pass through i, j. To limit the number of S-W cells computed, NCBI BLASTP uses an *X-drop* heuristic, described in [4], that terminates the recurrence at cells whose score $M_{i,j}$ is much lower than the best alignment score found so far.

Figure 2(a) shows the portion of the full S-W recurrence computed by a typical NCBI BLASTP gapped extension computation. Each pixel within the box represents one cell computed by the recurrence. Where full S-W would compute every cell in the box, NCBI BLASTP computes only the smaller darkened area, centered on the seed match (shown in white). BLASTP's restriction of S-W yields large savings in computation time in practice. However, the set of cells computed by the heuristic varies in size and is quite irregular in shape. Such variability is easily supported in software but is more challenging for a hardware implementation.



(a) NCBI BLASTP (b) Mercury BLASTP

Fig. 2. Typical structure of gapped extension in (a) NCBI and (b) Mercury BLASTP. X- and Y-axes indicate position within query and subject proteins. Cells computed by each method are shaded, with seed match in white.

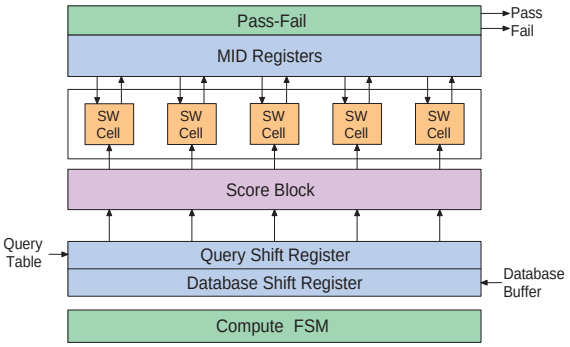


Fig. 3. Design of banded Smith-Waterman core with $\omega = 5$.

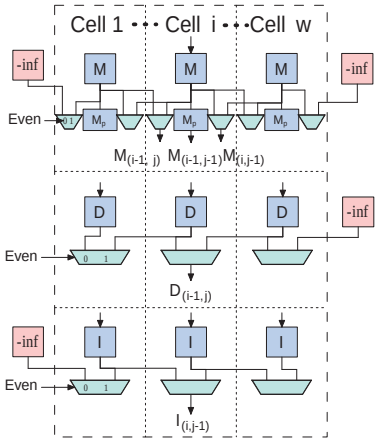


Fig. 4. Design of MID register block.

3. MERCURY BLASTP GAPPED EXTENSION

To avoid the complexity associated with NCBI BLASTP's irregular computation pattern, our accelerator uses an alternate strategy, the *banded Smith-Waterman* algorithm. This strategy was originally developed as an alternative software implementation, which is found in e.g. WU-BLAST [9]. The cells to be computed are limited *a priori* to a fixed-size band of anti-diagonal strips centered on the input HSP, as shown in Figure 2(b). The geometry of the band is defined by two parameters: the *band length* λ, which is the number of anti-diagonals computed in the band, and the *band width* ω, which is the number of cells computed on each anti-diagonal. It can be shown that this band covers exactly $\omega + \frac{\lambda}{2}$ residues in each of the two sequences.

Computation proceeds along anti-diagonals in a stair-step fashion from left to right in the band. To ensure that the optimal alignment found in the band is associated with the HSP that defined it, we impose the constraint that the alignment must cross the anti-diagonal at the center of the HSP. This constraint is implemented by not clamping the alignment score to zero once the center is crossed, thus forbidding the alignment to start anew, and by returning the best alignment score observed *after* crossing the center.

3.1. Banded Smith-Waterman core

Computation of the cells on each anti-diagonal is handled in our design by the banded S-W core. This core is implemented as a standard systolic array that computes ω cells of each anti-diagonal in parallel. Figure 3 shows its main components: the S-W cell array, the MID register block (retaining the recurrence values M, I, and D), the score block, the database-query shift register, and the pass-fail block.

Each cell in the systolic array implements logic to compute the recurrence for a single S-W cell. It consists of four adders, five maximizers, and a two-input mux to clamp scores either to zero, for cells before the HSP's center, or to negative infinity, for cells after the center.

The cell values computed by the array are stored in the *MID register block*. Because only the score of the optimal alignment is computed and not the alignment itself, only the two most recent anti-diagonals need be stored. As shown in Figure 4, four registers in each cell store the M, I, and D values computed in the previous clock cycle and the M value computed two cycles prior. Some of the local dependencies of a cell vary according to whether it is on an odd or even anti-diagonal (the leftmost anti-diagonal is odd). For odd cells, $M_{i-1,j}$ and $I_{i,j-1}$ are retrieved from its left neighbor, while for even cells $M_{i,j-1}$ and $D_{i-1,j}$ are retrieved from its right neighbor. Multiplexers are used in the MID register block to resolve these dependencies.

The *score block* generates a signed 8-bit δ score for each residue pair considered by a cell. δ is stored as a table in on-chip block RAM for single cycle access and is addressed by a concatenated residue pair. On our FPGA, each block RAM provides two independent read ports; we therefore use $\omega/2$ block RAMs to service the systolic cell array.

To perform gapped extension on a pair of sequences, the entire query is first loaded into on-chip block RAMs, after which the subject sequence is streamed in through a circular buffer. The active query and database residues are stored in a pair of parallel-tap *shift registers*, whose values are read by the score block. Residues are shifted in one per clock cycle, during the computation of odd anti-diagonals for the subject and even anti-diagonals for the query.

The *pass-fail block* simultaneously compares the ω cell scores in an anti-diagonal against a threshold. If any cell value exceeds the threshold, the HSP is deemed significant and is immediately passed through to software. We implement the following optimization to terminate extension early in some cases. Once an alignment crosses the HSP, its score is never clamped to zero but may become negative. If we observe only negative scores in all cells on two consecutive anti-diagonals, extension is terminated.

3.2. Supporting packed query sequences

Protein sequences are typically only about 300 residues long. However, the upstream stages of Mercury BLASTP can support query sequences that are significantly larger. It is therefore advantageous to concatenate several small query sequences into one composite query that is compared to the database in a single pass. Such *query packing* reduces the number of passes over the database and hence the overall search time.

Our gapped extension hardware uses *threshold* and *start* tables to support packed query sequences. In BLASTP, the score threshold to pass on a gapped alignment varies based on the query sequence. The threshold table maps any position in a packed query to the threshold score corresponding to the component query sequence at that location. The start table is used to identify the start of the current component at any point in the packed query. Knowing this start permits extension to begin at the start of an HSP's component, rather than considering the full composite query.

The worst-case running time per HSP is exactly $5 + \omega + \lambda$ clock cycles (5 to compute band geometry and initialize the database buffer, ω to load the shift registers with initial residues, and λ for the score computation). The use of the start table along with optimizations to support early alignment termination provide an average running time savings of 56% for $\omega = 65$ and $\lambda = 1601$ on typical protein datasets.

4. RESULTS

We have implemented Mercury BLASTP on the Mercury system [10], a disk-based, high-throughput architecture for reconfigurable logic. The system's host machine contains two 2.0 GHz AMD Opteron CPUs with 8 GB of memory, running Linux, and two prototyping co-processor boards connected via the PCI-X bus to the host. Interfacing drivers to the boards are provided by Exegy, Inc.[2] The first board contains a Xilinx Virtex-II 6000 FPGA (used for BLASTP stages 1 & 2); the second, a Xilinx Virtex-II 4000 FPGA (used for stage 3, the gapped extension stage). For an ω range of 35 to 75 anti-diagonals, the stage 3 design uses 33-50% of the slices and 45-62% of the block RAMs.

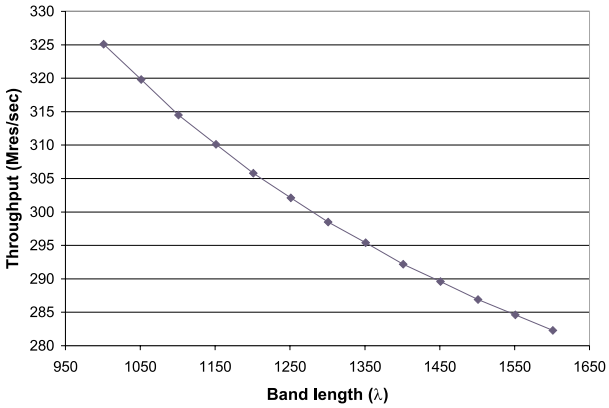


Fig. 5. Stage 3 throughput as a function of band length λ.

We augmented the NCBI BLASTP codebase to interface with our FPGA accelerator while preserving the original user interface, command-line options, and input/output formats. Queries are packed into 2048-residue bins, and the database is streamed through the three hardware filtering stages for each bin. Results from the hardware are passed to the host CPU for full gapped extension in software. In this configuration, we have demonstrated sustained data throughput from disk to FPGA well over the requirement for Mercury BLASTP (i.e., we are not I/O limited).

4.1. Performance of gapped extension

Stage 3 throughput and sensitivity is dependent on band geometry. The throughput of stage 3 is expressed as millions of database residues processed per second (Mres/sec). The throughput of this stage, as estimated by a simple mean-value model, is directly proportional to the design clock frequency and inversely proportional to the average HSP processing time and to the filtering rates of the initial stages of the BLASTP pipeline. Figure 5 plots the estimated stage 3 throughput as a function of band length, for a fixed width of 65. The largest estimated throughput for gapped extension is 325 Mres/sec for $\lambda = 1001$. Larger band geometries require more processing time per HSP and so reduce throughput. Our final design uses parameters $\omega = 65$ and $\lambda = 1601$ with a maximum estimated throughput of 282 Mres/sec.

[2] http://www.exegy.com/

Table 1. Execution time of Mercury BLASTP compared to the baseline system.

Experiment	Baseline Time	Mercury Time	Speedup
1	2,515 min	148 min	16.99×
2	3,168 min	225 min	14.08×
3	202 min	35 min	5.77×

Table 2. Sensitivity of Mercury BLASTP compared to the baseline system.

Experiment	Sensitivity	Alignments Lost	New Alignments
1	99.36%	37,799	11,218
2	99.45%	21,334	54,426
3	98.53%	38,444	35,812

4.2. Overall Performance of Mercury BLASTP

To quantify the performance of the entire Mercury BLASTP pipeline, we compared it to NCBI BLASTP running on a modern, general-purpose workstation with a 3.0 GHz Pentium D CPU and 1.5 GB of RAM running 64-bit Linux. We used a recent version of NCBI BLASTP (release 2.2.15) built with all available compiler optimizations of gcc 3.4. BLASTP runs were performed single-threaded on one core of the CPU. BLASTP was run with an E-value threshold of 10^{-5} and default parameters otherwise. Recorded runtimes include query setup and the time spent in the three stages of the pipeline but do not include time spent formatting the final alignments for printing.

The three comparisons performed were as follows:

1. *E. coli* K12 proteome (1.35 Mres) vs. GenBank Non-Redundant (NR) database (1.39 Gres);
2. *B. thetaiotaomicron* proteome (1.85 Mres) vs. GenBank NR;
3. *Y. pestis* KIM proteome (1.27 Mres) vs. all other bacterial proteomes in GenBank (282 Mres).

Query sequences were filtered to remove low-complexity regions.

Tables 1 and 2 respectively show the speedup and the sensitivity of Mercury BLASTP relative to the software baseline for our experiments. Sensitivity is measured as the fraction of alignments from the baseline output that were also found by Mercury BLASTP. Alignments in the two outputs that overlap by more than 50% were considered to be the same. We also report the number of alignments found by the baseline but not by Mercury BLASTP ("Alignments Lost") and vice versa ("New Alignments").

Mercury BLASTP averages more than an order of magnitude faster than the software baseline for the three experiments, with larger databases giving greater speedups. Furthermore, 99% of all alignments found by NCBI BLASTP were also detected by our FPGA solution. Mercury BLASTP finds about as many new significant alignments as it loses existing ones, suggesting that observed differences in sensitivity are within the normal error margin.

5. CONCLUSION

This paper presents the design of a banded Smith-Waterman accelerator for use within the context of the BLASTP application. This banded Smith-Waterman accelerator is distinct from the classical algorithm in that it performs alignments only within a bounded region around an HSP (provided by previous stages in the BLASTP application).

Overall, Mercury BLASTP is shown to execute between 6 and 16 times faster than NCBI BLASTP running on a modern processor, maintaining 99% sensitivity. These results are measured using experimental runs that are indicative of BLASTP usage common in the biological research community.

6. REFERENCES

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–97, Mar. 1981.

[2] D. T. Hoang, "Searching genetic databases on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 185–191.

[3] Y. Yamaguchi *et al.*, "High speed homology search with FPGAs," in *Pacific Symp. on Biocomputing*, 2002, pp. 271–282.

[4] S. F. Altschul *et al.*, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, pp. 3389–402, 1997.

[5] A. Jacob *et al.*, "FPGA-accelerated seed generation in Mercury BLASTP," in *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2007.

[6] J. Lancaster *et al.*, "Acceleration of ungapped extension in Mercury BLAST," *Int'l J. of Embedded Sys.*, 2007, in press.

[7] E. Sotiriades, C. Kozanitis, and A. Dollas, "Some initial results on hardware BLAST acceleration with a reconfigurable architecture," in *HiCOMB Workshop*, April 2006.

[8] M. Herbordt *et al.*, "Single pass, BLAST-like approximate string matching on FPGAs." in *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2006.

[9] S. F. Altschul and W. Gish, "Local alignment statistics," *Methods: a Companion to Methods in Enzymology*, vol. 266, pp. 460–80, 1996.

[10] R. D. Chamberlain *et al.*, "The *Mercury* system: Exploiting truly fast hardware for data search," in *Int'l Workshop on Storage Network Architecture and Parallel I/Os*, 2003.