Rolling Partial Prefix-Sums To Speedup Evaluation of Uniform and Affine Recurrence Equations

Narayan Ganesan[1], Roger D. Chamberlain[2], Jeremy Buhler[2] and Michela Taufer[1]

[1]Computer and Information Sciences, University of Delaware
[2]Computer Science and Engineering, Washington University in St. Louis

ABSTRACT

As multithreaded and reconfigurable logic architectures play an increasing role in high-performance computing (HPC), the scientific community is in need for new programming models for efficiently mapping existing applications to the new parallel platforms. In this paper, we show how we can effectively exploit tightly coupled fine-grained parallelism in architectures such as GPU and FPGA to speedup applications described by uniform recurrence equations. We introduce the concept of rolling partial-prefix sums to dynamically keep track of and resolve multiple dependencies without having to evaluate intermediary values. Rolling partial-prefix sums are applicable in low-latency evaluation of dynamic programming problems expressed as uniform or affine equations. To assess our approach, we consider two common problems in computational biology, hidden Markov models (HMMER) for protein motif finding and the Smith-Waterman algorithm. We present a platform independent, linear time solution to HMMER, which is traditionally solved in bilinear time, and a platform independent, sub-linear time solution to Smith-Waterman, which is normally solved in linear time.

Keywords: Dynamic Programming, HMMER, Protein-Motif Finding, GPUs, Parallelization, Computational Biology

1. GENERAL ROLLING PARTIAL PREFIX-SUMS ALGORITHM

Let D be a finite domain of points. Each point can corresponds to a unique sub-problem or cell in a dynamic programming matrix. Let F be a function from D to a "result" domain Σ (e.g., the real numbers) that corresponds to the computation of the cost point in D. We seek to compute the values $F(d)$ for a point $d \in D$. Let $(\Sigma, \wedge)$ form a commutative semigroup, i.e., the operator $\wedge$ is a commutative, associative binary operator on results.

Suppose that $F(d)$ is computable for any $d \in D$ as follows:

$$F(d) = \bigwedge(f_1(d), f_2(d), \ldots, f_n(d)) \tag{1}$$

where the summary $\bigwedge$ is the natural extension of $\wedge$ from two to any nonzero number of arguments. The summary operator maps two or more values in the results domain into a single value in the same domain. The function $f_i(d)$ is a mapping from multiple (finite number of) points in the domain D to one element in the results domain Σ. Here we consider only monadic recurrences where the function can be written as follows:

$$f_i(d) = F(d') \oplus h_i(d) \tag{2}$$

where $\oplus$ is a binary extension operator on the results $F(d')$ and $h_i(d) \in \Sigma$ is a "local" function that depends only on d, such as a look-up table and can be computed without the knowledge of any $F(d)$. The relation $d' < d$ must be satisfied, according to a partial order $<$, in order to avoid cyclic dependencies. The minimal elements of the partial order are "base" cases. A subset B of D is said to be "sufficient" for d if every path of dependency from d back to the base cases passes through an element of B. The nature of this dependency imposes a sequential execution of function F as dictated by the partial order. Therefore the number of algorithmic time-steps for sequential execution grows as the size of domain $D\,modulo\,<$, i.e., equal to total number of sets in D such that any two elements from different sets follow the partial order but not any two elements within the same set. For many problems this can grow significantly as the product of the input sizes.

E-mail: ganesan@udel.edu, roger@wustl.edu, jbuhler@wustl.edu, taufer@udel.edu

In this paper, we introduce the technique of rolling partial prefix-sum to extract parallel evaluations in recurrence equations such as in Equation 1. Our approach extends the prefix-sum algorithm by dynamically keeping track of and resolves multiple dependencies without having to evaluate the intermediary values. More specifically, we expand the prefix-sum approach to recurrences defined on a semiring and we introduce the technique of rolling partial-prefix sums for the same in order to extract parallel evaluations of the recurrence equations. This technique dynamically keeps track of and resolves the dependencies without evaluating the intermediary values. Our acceleration is applicable to the general framework of recurrence equations defined on a semiring. A few common examples of the semirings encountered in the context of dynamic programming are as follows:

$$\begin{array}{cccccccccc} \Sigma & — & \mathbb{R} & \mathbb{R} & \mathbb{R} & \mathbb{R} & \mathbb{R} & \mathbb{R} & \mathbb{R} \\ \wedge & — & + & \min & \min & \max & \max & \max & \min \\ \oplus & — & \times & + & \times & + & \times & \min & \max \end{array}$$

The general algorithm for rolling partial prefix-sums consists of five stages, executed in this order:

1. *Confirm the monadic and semiring nature of the recurrence equations for a problem* - This is important for reducibility of the recurrence equations and partial prefix-sum calculation.
2. *Identify the number of "sufficient" cases for each cell* - This is important to compute and maintain dependency information.
3. *Design the shape of wavefront(s)* - This is important for space requirement. The number of elements in the wavefront(s) as well as the number of dependencies for each element dictate the total space/memory requirement.
4. *Identify the direction of propagations for each wavefront* - This is important to ensure that the dependencies are preserved or the net change in the number of dependencies is non-positive, for a stable computational requirement. Each wavefront can propagate in an independent fashion.
5. *Design the systolic array for implementation*

We also describe how our technique can expose parallelism in two relevant applications in bioinformatics, the popular HMMER[1,2] program for protein motif finding (Section 2) and the Smith-Waterman algorithm[3] for sequence alignment in (Section 3). The recurrence equations in the two applications satisfy the above semiring requirement and thus our technique of rolling partial prefix-sums can be applied to accelerate their evaluation on multithreaded and reconfigurable logic architectures. In general, our technique is applicable to problems that fall in the general framework described above and is platform independent. The paper concludes by summarizing our contributions and discussing related work (Section 4).

2. APPLICATION TO PLAN 7 HMM

2.1 Recurrences in Plan 7 HMM

Hidden Markov Model (HMMs)[2,4] are structured according to the Plan 7 schema. In this schema, a HMM of length m (e.g., $m = 5$ in Figure 1) contains m "match states" $M_1 ... M_m$. A parallel sequence of "deletion states" states $D_2 ... D_{m-1}$ allows any substring of sequence positions to be skipped, while another parallel set of "insertion states" $I_1 ... I_{m-1}$ allows for substring insertions between any two sequence positions. The states B and E act as the HMM's initial and final states in Figure 1. The feedback loop through state J allows for output sequences to be repeated. The decoding of a Plan 7 Hidden Markov Model via the Viterbi algorithm yields the maximum probability of the Hidden Markov Model emitting the observed protein sequence. The value $V(i, j)$ so obtained gives the best cost of aligning (the maximum probability of observing) the first i symbols of the protein sequence to the first j states of the HMM. The calculations are performed in the log domain which eliminates floating point multiplications. The cost V can be further broken down into V_M, V_I, and V_D, i.e., the cost of aligning the first i symbols of the sequence x to the jth match, insertion, and deletion states respectively.

$$V_M(i,j) = \epsilon_M(x_i, j) + \max \begin{cases} V_M(i-1, j-1) + T(j-1, c_1), \\ V_I(i-1, j-1) + T(j-1, c_2), \\ V_D(i-1, j-1) + T(j-1, c_3), \\ B_{i-1} + T(j, c_4) \end{cases} \quad (3)$$

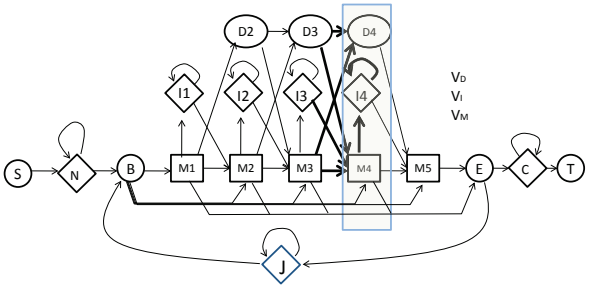

Figure 1. Plan 7 profile hidden Markov model of length $m = 5$.

$$V_I(i,j) = \epsilon_I(x_i, j) + \max \begin{cases} V_M(i-1,j) + T(j,c_5), \\ V_I(i-1,j) + T(j,c_6) \end{cases} \tag{4}$$

$$V_D(i,j) = \max \begin{cases} V_M(i,j-1) + T(j,c_7), \\ V_D(i,j-1) + T(j,c_8) \end{cases} \tag{5}$$

For the above equations, it can be seen that the operator $\bigwedge$ is $\max$ and $\oplus$ is $+$. Regardless of whether the calculations are performed in probability or log probability domain, recurrence equations described by the associative operators $\max$ and $+$ or $\times$ satisfy the semiring property and are amenable to this technique. Here, T is the look-up table containing the HMM transition probabilities and $c_1, \cdots, c_8$ are constant indices that denote the transitions between $M_{j-1} \rightarrow M_j, I_{j-1} \rightarrow M_j, D_{j-1} \rightarrow M_j, B \rightarrow M_j, M_j \rightarrow I_j, I_j \rightarrow I_j, M_{j-1} \rightarrow D_j$ and $D_{j-1} \rightarrow D_j$ respectively. The corresponding row $j-1$ holds the transition data between HMM's (match, insertion, and deletion) states at the $j-1$th and jth positions along with the transition to the beginning and end states B and E. The table ϵ_M and ϵ_I which is indexed by the sequence symbol x_i and the HMM state j, stores the corresponding emission probability for the symbol x_i for the match state M_j and insertion state I_j respectively. It can be seen that $h(.)$ defined in Equation 2 for the current recurrence is $T(...) + \epsilon$. Thus the table needs to be accessed every iteration in order to compute the match, insertion, and deletion costs. The costs V_M and V_I for row i depend only on the corresponding values V_M, V_I and V_D for previous row $i-1$. The term B_{i-1}, which is the cost of aligning the first $i-1$ input symbols to a sequence of HMM states ending in the beginning state B, depends only on costs from the previous row. This value is determined only after the knowledge of all the cells in row $i-1$ and thus it imposes a computational dependency on the last cell in row $i-1$ - after the knowledge of which, the value of B_{i-1} can be computed (see Figure 2). The presence of this term in the recurrence equation is characteristic to HMMER recurrence equations.

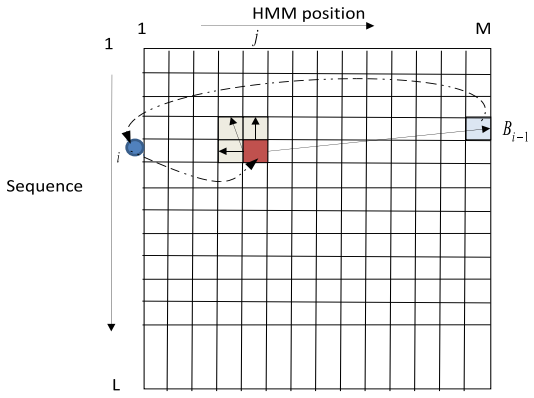

Figure 2. Cell (i,j) depends on $(i-1,j-1), (i,j-1), (i-1,j)$ and $(i-1,M)$. The dotted arrows indicate HMM dependencies and solid arrows denote computational dependencies.

This dependence on last cell from previous row entails a row major order of computation of the dynamic programming matrix and thus eludes many parallelization techniques. The dependency of $V_D(i,j)$ on the previous cell $V_D(i,j-1)$ also entails a serial evaluation of each row which makes computational complexity of the system $O(M \times L)$ for the dimensions of the matrix M and L respectively. The memory requirement is $O(M)$ to the store costs corresponding to a single row.

2.2 Sufficient Cases For Dependencies

The key for exposing parallelism in problems based on recurrence equations is to identify the dependencies and resolve them dynamically throughout the execution. In this section, we examine the dependency for a particular cell $(2,2)$ and move to a general case (i,j). From the recurrence Equations 3-5:

$$V_M(2,2) = \epsilon_M(x_2,2) + \max \left\{ \begin{array}{ll} V_M(1,1) + T(1,c_1), & V_I(1,1) + T(1,c_2) \\ V_D(1,1) + T(1,c_3), & B_1 + T(1,c_4) \end{array} \right.$$

With the values of costs from cell $(1,1)$ known, the expression for $V_M(2,2)$ can collapse to:

$$V_M(2,2) = \max(m_{22}, B_1 + m^1_{22}) \tag{6}$$

where

$$m_{22} = \epsilon_M(x_2,2) + \max \left\{ \begin{array}{ll} V_M(1,1) + T(1,c_1), & V_I(1,1) + T(1,c_2) \\ V_D(1,1) + T(1,c_3) & \end{array} \right. \tag{7}$$

and $m^1_{22} = \epsilon_M(x_2,2) + T(1,c_4)$. The cost of V_M for any cell along the second row can be written as $V_M(2,j) = \max(m^0_{2j}, B_1 + m^1_{2j})$ where the cost depends on the value of B_1. Similarly, for any cell in the third row, the cost V_M depends on the values of B_1 and B_2. Note that for any cell in row i, the corresponding costs V_M, V_I, and V_D depend on values $B_1, \cdots, B_{i-1}$ only. By a mathematical induction argument, the cost of V_M for any column j in row i can be written as:

$$V_M(i,j) = \max(m^0_{ij}, B_1 + m^1_{ij}, \cdots, B_{i-1} + m^{i-1}_{ij}) \tag{8}$$

Similarly, the cost of V_I for any cell (i,j) can be written as:

$$V_I(i,j) = \max(a^0_{ij}, B_1 + a^1_{ij}, \cdots, B_{i-1} + a^{i-1}_{ij}) \tag{9}$$

Finally, V_D for any cell (i,j) can be written as:

$$V_D(i,j) = \max(d^0_{ij}, B_1 + d^1_{ij}, \cdots, B_{i-1} + d^{i-1}_{ij}) \tag{10}$$

for some numerical values m^k_{ij}, a^k_{ij}, and d^k_{ij} that describe the dependency of the costs on the value of B_k. The numerical values m^0_{ij}, a^0_{ij}, and d^0_{ij} denote the independence of the costs. Therefore the chain of dependencies from cell (i,j) can be found to pass through the sufficient cases with values $B_1, \cdots, B_{i-1}$.

2.3 Wavefront Design and Propagation

With the above expressions in place, it is possible to carry only the dependency information from the cell (i,j) to the dependent cells $(i+1,j), (i,j+1)$, and $(i+1,j+1)$ without having to know any of the values of $B_1, \cdots, B_{i-1}$. By substituting the costs given by Equations 8, 9, and 10 into the recurrence Equations 3, 4, and 5 and by comparing the dependencies on either side of the equation, we get:

$$m^k_{i,j} = \epsilon_M(x_i,j) + \max \left\{ \begin{array}{l} m^k_{i-1,j-1} + T(j-1,c_1), \\ a^k_{i-1,j-1} + T(j-1,c_2), \\ d^k_{i-1,j-1} + T(j-1,c_3) \end{array} \right. \tag{11}$$

$$a^k_{i,j} = \epsilon_I(x_i,j) + \max \left\{ \begin{array}{l} m^k_{i-1,j} + T(j,c_5), \\ a^k_{i-1,j} + T(j,c_6), \end{array} \right. \tag{12}$$

and

$$d^k_{ij} = \max \left\{ \begin{array}{l} m^k_{i,j-1} + T(j,c_7), \\ d^k_{i,j-1} + T(j,c_8) \end{array} \right. \tag{13}$$

where $k = 0, 1, \cdots, i-1$. Finally, with the actual numerical value of a particular B_k known (where B_k is determined from B_{k-1} and the values of m^k_{kj}), the dependency can be resolved by absorbing it into the independent value m^0_{ij} as follows:

$$m^0_{ij} \leftarrow \max(m^0_{ij}, B_k + m^k_{ij}) \tag{14}$$

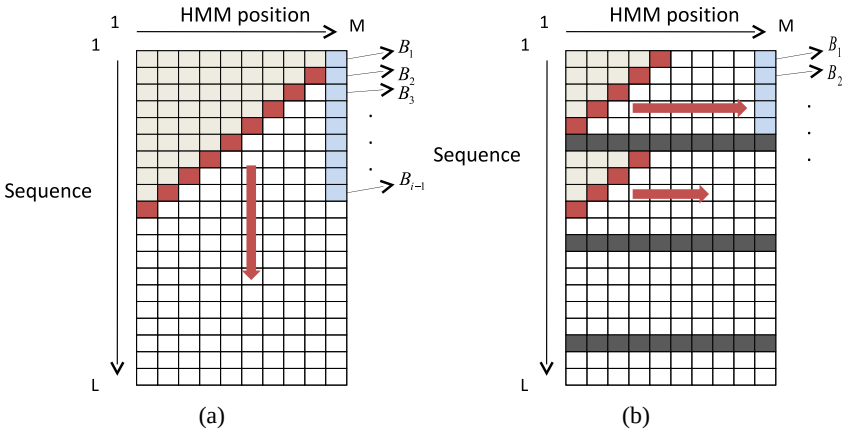

Figure 3. (a) The elements can be computed as in the case of Smith-Waterman along the minor diagonal all at once. Once the numerical value of B_i is known, it is propagated to all the cells to resolve their dependency on B_i. (b) An alternative strategy for memory/area constrained systems.

similarly:

$$a_{ij}^0 \leftarrow \max(a_{ij}^0, B_k + a_{ij}^k) \tag{15}$$

$$d_{ij}^0 \leftarrow \max(d_{ij}^0, B_k + d_{ij}^k) \tag{16}$$

Hence, the above recurrence equations for m_{ij}^k, a_{ij}^k, and d_{ij}^k contain only local dependencies in contrast to the original recurrence equation for the costs (see Equation 3). The wavefront for evaluating Equations 17, 12, and 13, which follow only local dependencies, is computed concurrently along the anti-diagonal as shown in Figure 3(a).

If $L > M$ (i.e., the sequence length is greater than the HMM length (number of match states of the HMM) then the wavefront is propagated along the sequence, with the arrival of each new symbol as shown in Figure 3(a). At time-step M, there are $M-1$ dependencies $B_1, \cdots, B_{M-1}$ to keep track of. At the same instant, the top of the wavefront passes through the cell $(1, M)$ at which point the value of B_1 is determined. This information can be used to resolve the dependency of all elements on B_1 concurrently via Equations 14, 15, and 16. Similarly at the next time-step, an additional dependency B_M is encountered and dependency on B_2 is resolved concurrently. Following the same pattern, at any time step the elements in wavefront depend only on the previous $M-2$ values, whose dependencies can all be dynamically calculated and resolved throughout the computation as described above. Hence, the name *rolling* partial prefix-sums. If $L < M$ (i.e., the HMM length is greater than the sequence), then the wavefront is propagated along the model axis. The elements in the wavefront now have static dependencies which is preserved throughout the course of computation.

2.4 Implementation via Systolic Array

The computation described above is implemented via a systolic array design. Since at any time step i, the entire wavefront of length M cells depends on utmost $M-2$ values each, i.e., $B_{i-1}, \cdots, B_{i-M+2}$, the memory requirement is bound to be $O(M^2)$. The exact requirement is calculated by the sum $\sum_{i=1}^{M-2} i = (M-2)(M-1)/2$, since each cell in the wavefront has different number of dependencies. The layout of the systolic array is shown in Figure 4 where each row of the array computes and updates dependencies of one cell of the wavefront as marked. This is realized by a triangular array and the dependencies are shifted right each time step to make room for the newest dependencies. The value of B_{i-M+2} is computed by one element of the array represented by the unshaded block which is then used to resolve the dependencies of all the cells. At every successive iteration, update dependency information following Equations (17-13), resolves any dependency via Equations (14-16); shift right operations are performed concurrently to make room for any new dependency. The size of the systolic array is determined by the size (length) of the wavefront. If $L > M$, the maximum length of wavefront is M and takes $(M-1)(M-2)/2$ elements to maintain and update the rolling dependency information. The time complexity is given by the time for the wavefront to sweep through the entire dynamic programming matrix and is bound by the sum of HMM and sequence lengths $O(M+L)$.

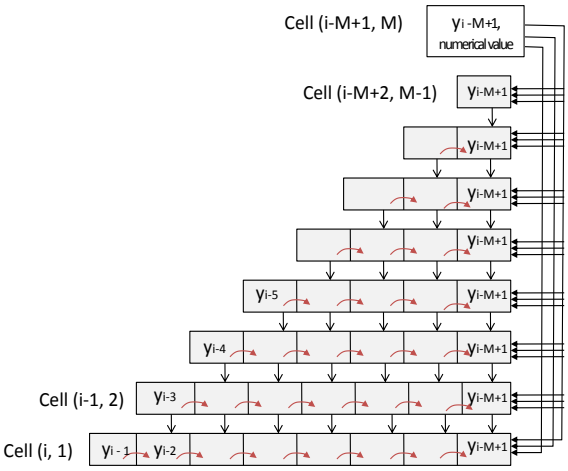

Figure 4. Systolic array design of parallel Plan 7 Viterbi decoding. The shaded elements hold and update the dependency on the variables they are currently labeled by. At the next iteration, the cells are shifted right to make room for the new dependencies.

A HMM of approximately 700 positions (match states) requires $\sim$ 245,000 concurrent updates, depending on representation of cost values. The current availability of shared memory resources in devices such as GPUs mandates the redesign of some aspects of our algorithm. By tiling the dynamic programming matrix, shared memory requirements can be relaxed. In order to limit the size of the wavefront, the matrix is tiled along the model axis as shown in the Figure 3(b). Depending on the height of the tiles s, the wavefront can be made to satisfy the memory constraints. This works by processing input symbols in batches, by propagating the wavefront along the entire model axis, and by proceeding to the next batch. Since the dependencies are static for each tile, the dynamic resolution of dependencies is not required. The boundary between the tiles is resolved before the next input batch is processed. This requires two passes through each tile, first to evaluate the sufficient cases $B_1, \cdots, B_s$ and second to compute the boundary values. Thus the total time for each tile is $2(M + s)$ with exactly L/s tiles to process. The total time taken to decode the sequence is $2(M + s)L/s$. The above algorithm is still bilinear but with a speedup of $s/2$ as opposed to a fully linear implementation. The speedup depends on the size of the batch processed s and the memory requirement is $(s - 1)(s - 2)/2$.

3. APPLICATION TO SMITH-WATERMAN

3.1 Recurrences in Smith-Waterman

The recurrence equations for the Smith-Waterman[3] algorithm with affine gap penalties and substitution costs are:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + T(x_i, y_j), \\ I_x(i-1,j-1) + T(x_i, y_j), \\ I_y(i-1,j-1) + T(x_i, y_j) \end{cases} \tag{17}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - A, \\ I_x(i-1,j) - B \end{cases} \tag{18}$$

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - A, \\ I_y(i,j-1) - B \end{cases} \tag{19}$$

The sequences are denoted by x (database) and y (query) of lengths X and Y respectively. We can assume $X \gg Y$ without loss of generality. A and B are gap-open and gap-extent penalties and T is the table of substitution costs. It can be seen that the above recurrence equations resemble the HMMER equations except for the term involving B_i which is missing in Smith-Waterman. Therefore the dependencies in this application are reduced to simple local dependencies. The equations are traditionally evaluated by propagating the minor-diagonal along the database or query axis in a fine-grained parallel architecture, thus taking $O(X + Y)$ time to evaluate the final alignment cost $M(Y, X)$. In our approach, we take advantage of local dependencies in the recurrence equations to evaluate the final alignment cost in sub-linear time without any heuristics.

3.2 Sufficient Cases For Dependencies

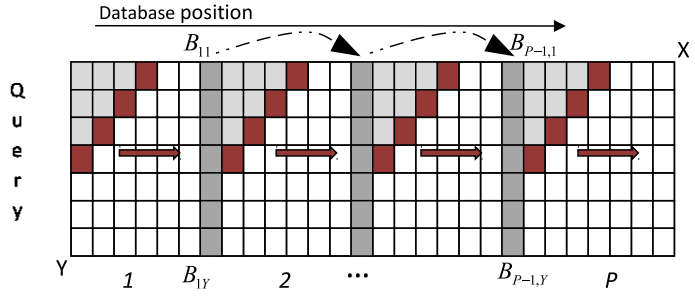

Figure 5. Partitioning of the Smith-Waterman dynamic programming matrix into P partitions to speedup the evaluation.

As shown in Figure 5, we partition the entire dynamic programming matrix into P partitions with length of each partition X/P. The sufficient cases (dependencies) for any cell (i, j) in partition p are $B_{p-1,1}, \cdots, B_{p-1,i}$ with utmost Y sufficient cases (dependencies) for any cell in the entire matrix. If we assume that $Y < X/P$, this case study is similar to the HMMER application with sequence shorter than the HMM length ($L < M$). Each partition statically depends on utmost Y values, thus eliminating the need for rolling dynamic dependency resolution.

3.3 Wavefront Propagation

By choosing P wavefronts along the anti-diagonal for each partition, a dependency relation is built between the sufficient cases B_{ij} and $B_{i+1,j}$ via a relation akin to Equations 8-10 in $X/P + Y$ time-steps. With the knowledge of $B_{11}, \cdots, B_{1Y}$ determined after $X/P + Y$ time-steps and the dependency information relating B_{ij} to $B_{i+1,j}$ known, the dependencies are resolved via equations identical to Equations 14-16. The total latency to propagate the dependencies along all the partitions is P. Thus the total time to determining the final alignment cost $M(Y, X)$ is $X/P+P+Y$. The optimal number of partitions P is $\sqrt{X}$ and the optimal time is $2\sqrt{X} + Y$. If Y is comparable to X, the query axis can also be partitioned similarly to accelerate the evaluation.

3.4 Systolic Array

Due to the similarity of recurrence equations to the HMMER application, the systolic array follows the same design. Since each wavefront is implemented as a different systolic array of size $(Y - 1)(Y - 2)/2$, the total size requirement is $O(P \times Y^2)$. The size required for the minimum latency systolic array is $\sqrt{X}(Y - 1)(Y - 2)/2$.

4. DISCUSSION AND RELATED WORK

In this paper we expand the prefix-sum approach to expose parallelism for recurrences defined on a semiring. The prefix-sum algorithm is traditionally applied to simple associative operators.[5] We extend this algorithm by introducing the concept of rolling partial prefix-sums that allows us to dynamically resolve dependencies at run-time. We present two applications of our approach to dynamic programming problems (i.e., the Plan 7 HMM and Smith-Waterman) for which we use rolling partial prefix-sums to decrease the time-complexity of the evaluations supported by the most suitable choice of wavefront and propagation direction. In particular, the time complexity for HMMER recurrences decreases from $O(M \times L)$ to $O(M + L)$ with the additional space requirement from $O(M)$ to $O(M^2)$. Similarly, the Smith-Waterman or edit-distance type recurrences can be solved in $O(2\sqrt{X} + Y)$ time; traditionally it is solved in $O(X + Y)$. Memory requirement for Smith-Waterman becomes $O(\sqrt{X} \times Y^2)$ for $X \gg Y$ - it is normally $O(Y)$. A successful application of this technique for HMMER recurrence is presented in previous work of the authors[6] wherein the evaluation of the dynamic programming matrix is parallelized in one dimension along each row. This implementation is mapped to the GPU architecture due to the similarity of the arithmetic operations and uniformity of memory accesses. A significant speedup is obtained compared to other popular implementation on the same architecture.[7]

The novelty of our work is in the proposed algorithmic improvement. Prefix-sums have been applied to bio-sequence comparisons[8] for parallelizing evaluations of rows. Automatic generation of systolic arrays to implement pipelined evaluation of uniform and affine recurrence equations[9] have been studied and found to give significant speedup for many

problems. Numerous accelerators for HMMER and Smith-Waterman exist for which the evaluation time is decreased via a combination of architectural improvements,[8,10–12] data path redesign,[13–18] and heuristics.[19,20] To our best knowledge, none of the accelerators mentioned above do rely exclusively on algorithmic improvement techniques as we do. Thus our approach can be considered highly platform independent.

5. ACKNOWLEDGMENTS

This research was supported by NIH award HG003225, NSF awards DBI-0237902, ITR-0427794, #0941318, #0922657, the U.S. Army grant #ARO 54723-CS, the NVIDIA University Professor Partnership Program and Exegy, Inc. R.D. Chamberlain is a principal in Exegy, Inc.

REFERENCES

[1] Eddy, S., "Profile hidden markov models," *Bioinformatics* **14**, 755–863 (1998).

[2] Eddy, S., "HMMER: Profile HMMs for protein sequence analysis." http://hmmer.janelia.org (2004).

[3] Smith, T. F. and Waterman, M. S., "Identification of common molecular subsequences," *J. Molecular Biology* **147**, 195–97 (mar 1981).

[4] Krogh, A. et al., "Hidden markov models in computational biology: Applications to protein modeling," *J. Molecular Biology* **235**, 1501–1531 (1994).

[5] Blelloch, G. E., "Prefix sums and their applications," in [*Synthesis of Parallel Algorithms*], Reif, J. H., ed., Morgan Kaufmann (1990).

[6] Ganesan, N., Chamberlain, R. D., Buhler, J., and Taufer, M., "Accelerating HMMER on GPUs by implementing hybrid data and task parallelism," in [*Proc. of the First ACM Int. Conf. on Bioinformatics and Computational Biology (ACM BCB)*], (2010).

[7] Walters, J. P., Balu, V., Kompalli, S., and Chaudhary, V., "Evaluating the use of GPUs in liver image segmentation and HMMER database searches," in [*Proc. of IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*], (2009).

[8] Aluru, S., Futamura, N., and Mehrotra, K., "Parallel biological sequence comparison using prefix computations," *J. Parallel and Distributed Computing* **63**(3), 264–272 (2003).

[9] Quinton, P., "The systematic design of systolic arrays," in [*Automata Networks in Computer Science: Theory and Applications*], 229–260, Princeton University Press (1987).

[10] Walters, J., Qudah, B., and Chaudhary, V., "Accelerating the HMMER sequence analysis suite using conventional processors," in [*Proceedings of AINA*], (2006).

[11] Hughey, R., "Parallel hardware for sequence comparison and alignment," *Comput. Appl. Biosci.* **12**, 473–479 (1996).

[12] Woznaik, A., "Using video-oriented instructions to speed up sequence comparison," *Comput. Appl. Biosci.* **13**, 145–150 (1997).

[13] Horn, D., Houston, M., and Hanrahan, P., "ClawHMMER: A streaming HMMER-search implementation," in [*Proc. of the ACM/IEEE Conference on Supercomputing (SC)*], (2005).

[14] Farrar, M., "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics* **23**, 156–161.

[15] Lindahl, E., "Altivec HMMER, version 2.3.2." http://powerdev.osuosl.org/project/hmmerAltivecGen2mod/.

[16] Rognes, T. and Seeberg, E., "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics* **16**(8), 699–706 (2000).

[17] Derrien, S. and Quinton, P., "Parallelizing HMMER for hardware acceleration on FPGAs," in [*Proc. of Application-specific Systems, Architectures and Processors Conf.*], 10–17 (jul 2007).

[18] Oliver, T., Yeow, L. Y., and Schmidt, B., "Integrating FPGA acceleration into HMMER," *Parallel Computing* **34**(11), 681–691 (2008).

[19] Maddimsetty, R., Buhler, J., Chamberlain, R., Franklin, M., and Harris, B., "Accelerator design for protein sequence HMM-search," in [*Proc. 20th ACM International Conference on Supercomputing*], 288–296 (2006).

[20] Jacob, A., Lancaster, J., Buhler, J., and Chamberlain, R., "Preliminary results in accelerating profile HMM search on FPGAs," in [*Proc. of Workshop on High Performance Computational Biology (HiCOMB)*], (2007).