Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism

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

[1] University of Delaware
Dept. of Computer & Inf. Sciences
Newark, DE 19716
{ganesan, taufer}@udel.edu

[2] Washington University in St. Louis
Computer Science and Engineering
St. Louis, MO 63130
roger, jbuhler@wustl.edu

ABSTRACT

Many biologically motivated problems are expressed as dynamic programming recurrences and are difficult to parallelize due to the intrinsic data dependencies in their algorithms. Therefore their solutions have been sped up using task level parallelism only. Emerging platforms such as GPUs are appealing parallel architectures for high-performance; at the same time they are a motivation to rethink the algorithms associated with these problems, to extract finer-grained parallelism such as data parallelism.

In this paper, we consider the *hmmersearch* program as a representative of these problems and we re-design its computational algorithm to extract data parallelism for a more efficient execution on emerging platforms, despite the fact that *hmmersearch* has data dependencies. Our approach outperforms other existing methods when searching a very large database of unsorted sequences on GPUs.

1. INTRODUCTION

Many of the biologically motivated problems such as Smith-Waterman and Viterbi decoding of profile and generalized hidden Markov models are expressed as dynamic programming problems. Dynamic programming poses the solution to these problems in terms of solution to sub-problems, thus giving rise to a system of recurrence equations. The nature of the relationship of the sub-problems to the original problems defines the type of dependency relations in the system of recurrence equations that have to be evaluated in order to find the optimal solution. The time complexity to find a solution depends on the size of the dataset considered. With the current size of biological databases and their rate of growth, the time order of the solution for biologically motivated problems grows rapidly becoming intractable even on supercomputers. Thus it is imperative that we accelerate the evaluation of the problem recurrence equations either via efficient heuristics or parallelization schemes or both. Heuristics enable fast searches for solutions for large sized problems but suffer from loss of sensitivity either due to approximations or simplifications of the recurrence relations involved. Parallelization techniques mainly use task parallelism to execute each single sub-problem on e.g., one node or core of a cluster, and multiple sub-problems are performed in parallel. So for example, in the P7Viterbi algorithm used in the *hmmersearch* program for protein motif finding, a single sequence is normally assigned to a process or thread [11], resolving in this way any dependencies. Unfortunately large sequences lead to sub-problems that are too large for fast execution on single nodes.

Ubiquity computing capabilities of emerging parallel architectures such as graphics processing units (GPUs), multicores, field-programmable gate array (FPGAs), and single instruction, multiple data (SIMD) architectures provide a strong motivation to combine task parallelism with data parallelism. However recurrence relations among these sub-problems are not trivial to parallelize in applications such as protein motif finding because of the nature of the dependencies involved. In the past, this has prevented the use of hybrid parallelism (i.e., task and data parallelism) in applications based on dynamic programming.

In this paper, we consider the *hmmersearch* program [3, 4] for acceleration. We re-design its computational algorithm to extract data parallelism out of the serializing data dependencies, for a more efficient execution on platforms such as GPUs. We present an algorithm which parallelizes and resolves the dependencies in the HMMER search by using a hybrid parallelization strategy (i.e., by combining task and data parallelism) and does not rely on heuristics to increase the speed-up. The speed-up is motivated by algorithmic improvements rather than architectural implementations. The acceleration is achieved by developing a symbolic relation along the chain of dependencies and resolving them using data parallelism. Our technique is amenable to a variety of implementations, including hardware (FPGA), multi-threaded processing (GPU), or SIMD engines, which benefit from architecture dependent optimizations such as aligned data access and local dependencies. More in particular, we propose and implement a strategy to parallelize evaluations of recurrence equations by partitioning the chain of dependencies in a uniform and regular fashion.

The rest of this paper is organized as follows: Section 2 gives

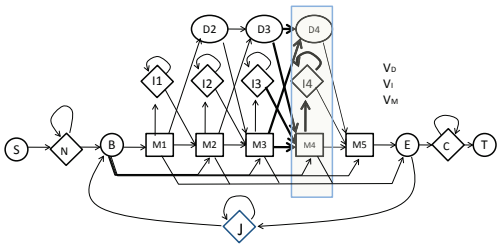

Figure 1: Plan 7 profile hidden Markov model of length 5.

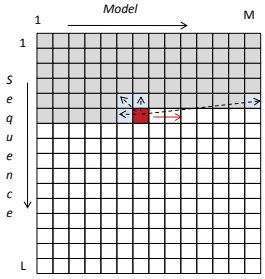

Figure 2: Sequential dependency along the row-index j in V_D.

and overview of the *hmmersearch* program, the GPU platform, and related work. Section 3 describes the parallelization technique to reduce the sequentially dependent computation and shows how to apply it to HMMER. Section 4 presents performance results of our techniques on GPU. Section 5 summarizes the current and future work.

2. BACKGROUND AND RELATED WORK

2.1 Viterbi Decoding of Plan7 HMM

Hidden Markov Models(HMMs) are used in computational biology [6] to group a family of related biological sequences and summarize their shared features using a probabilistic model. HMMs are structured according to the Plan 7 schema, as illustrated in the example in Figure 1. 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 sequence. The value so obtained, V, gives the best cost of aligning (the maximum probability of observing) the first i symbols of the output sequence to the first j states of the HMM. The cost V can be further broken down into V_M, V_I, and V_D, i.e., the cost of aligning the sequence to the jth match, of inserting, and of deleting states respectively. The costs, V_M and V_I for row i depend only on the corresponding values V_M, V_I, and V_D for the previous row $i-1$. The recurrence equation of interest in this application is the one governing the cost V_D as it exhibits the aforementioned sequential dependency along the row-index j (see Figure 2). A detailed description of the equations can be found at [3, 4].

2.2 GPU Architecture

The GPU architecture is designed to support high degree of parallelism in the form of large number (typically 1000s) of identical threads. This multithreaded architecture is extremely useful in concurrently executing many parallel identical tasks. The identical tasks are executed via the tightly coupled threads which communicate to each other via a hierarchy of shared memories. The CUDA platform enables the developer to program for a unified architecture supported by NVIDIA GPUs. The threads in the CUDA platform execute identical tasks and communicate via the fast but limited shared memory or the large and slower global memory. The parallelization technique described above is an excellent candidate for GPU implementation due to identical nature of operations involved. An in-depth discussion of GPU architecture and the CUDA platform can be found at [1].

2.3 Related Work

HMMER search was initially parallelized for clusters using MPI in [7]. Since the current match and insert states depend only on the previous row of match and insert states, maximum available parallelism was used to concurrently update the costs. In addition the state loop was also vectorized to process 24 HMM states in SIMD fashion or 8 state triplets at once. Implementations of HMMER search for graphics processing units (GPUs) were carried out in Claw-HMMER [5] and GPU-HMMER [11]. The challenge was to efficiently utilize the GPU resources to decode sequences independently using a task parallelism strategy, i.e., one sequence per thread. Optimizations included the use of GPU texture and shared memory to efficiently store and retrieve e.g., the calculated probabilities values. In addition to multiprocessor systems, a number of attempts to accelerate implementation of the HMMER recurrence have been carried out for FPGAs ([8, 9, 10]).

3. METHODOLOGY

3.1 Parallelization of Row Computations

In previous parallelizations [11, 5], each sequence of the dataset is assigned to a single thread (for GPUs) or a single process (for clusters). Decoding each sequence comprises evaluating a dynamic programming (DP) matrix of size $M \times L$, where M is the size of the model and L is the size of the sequence. Thus, decoding a single sequence of length L against a model of size M is of time complexity $O(ML)$. Figure 3 shows an example of this evaluations on GPU where each single sequence is assigned to a thread in a block.

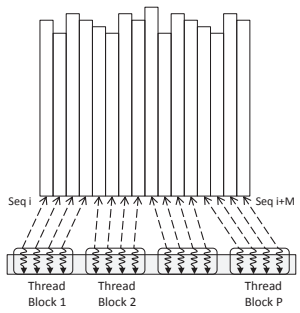

Figure 3: Task parallelism implemented on GPU.

In the methodology proposed in this paper, we reassign the

decoding of a single sequence across multiple threads to implement the data parallelism. Note that the data parallelism is built upon and extend the existing task parallelism. Figure 4 shows an example of the data parallelism on GPU where all the threads in a single block are assigned to decode a single sequence.

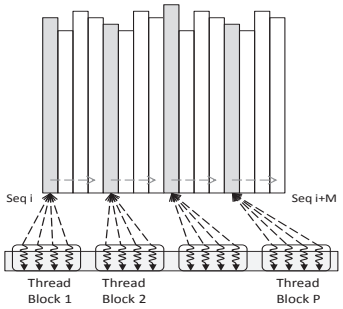

Figure 4: A hybrid of task and data parallelism for GPU.

Within the evaluation of a single sequence, we accelerate the computation of the matrix rows by partitioning each row into uniform set of contiguous cells and computing the dependencies between the partitions identically and independently in a data parallel setting. The parallel evaluation of the rows proceeds in three phases (Figure 5). In *Phase 1*, individual threads are employed to compute the relationship between the beginning and end of each partition. This enables the fast computation of boundary elements from the preceding boundary element to the left. Figure 5 shows the boundary elements between the partitions (shaded cells), also called anchor elements, that relate the costs between the beginning and end of the partition. For example, the first anchor element located at portion $k-1$ is related to **Cell 1**, the second anchor element located at portion $2k-1$ is related to **Cell k**, and so on. In *Phase 2*, by using the functions relating the boundary elements from the previous phase, we compute the numeric values for each of the boundary elements. These dependencies are represented by the dashed arcs in Figure 5. This phase must execute sequentially; with p partitions, it only requires p steps, rather than the N sequential steps that is required in the traditional approach. In *Phase 3*, by using the now up-to-date numerical values for each of the anchor elements, each partition independently computes the numeric values for all of the elements within the partitions in the data parallelism setting.

While Phases 2 and 3 are computation phases, Phase 1 is the critical phase that enable data parallelism by building the relationship among the different partitions. The equation embodying the sequential dependency can be written as:

$$V_D(i,j) = \max(V_D(i,j-1) + x_j, y_j) \tag{1}$$

where $V_D(i,j)$ is the delete state cost for Cell (i,j) and $x_j = T(j,c_7)$ and $y_j = V_M(i,j-1) + T(j,c_8)$ [3, 4].

The cost of the next cell $V_D(i,j+1)$ is expressed in terms of the cost Cell (i, j-1), i.e., $V_D(i,j-1)$, by substitution into

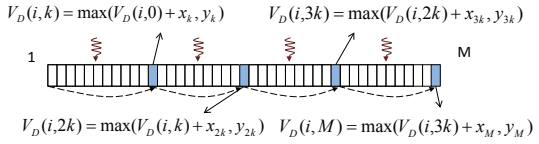

Figure 5: Parallel evaluation of matrix rows.

the previous relation as:

$$V_D(i,j+1) = \max(V_D(i,j-1) + x_{j+1}, y_{j+1}) \tag{2}$$

where $x_{j+1} = x_j + T(j+1,c_7)$ and

$$y_{j+1} = \max(y_j + T(j+1,c_7), V_M(i,j) + T(j+1,c_8)).$$

By extending this relation, we can rewrite the dependency relating costs $V_D(i,j+k)$ and $V_D(i,j-1)$ as follows:

$$V_D(i,j+k) = \max(V_D(i,j-1) + x_{j+k}, y_{j+k}) \tag{3}$$

where,

$$x_{j+k} = x_{j+k-1} + T(j+k-1,c_7) \tag{4}$$

and

$$y_{j+k} = \max \left\{ \begin{array}{c} y_{j+k-1} + T(j+k-1,c_7), \\ V_M(i,j+k-1) + T(j+k-1,c_8) \end{array} \right. \tag{5}$$

By the above procedure any two costs, i.e., any k number of cells apart, can be related by updating only Equations 4-5 in k iterations, even without the knowledge of the costs themselves. Here, the operator max is the primary binary associative operator over which the dependency information is computed. In general, this could also be computed in the presence of the secondary binary operator +, lookup tables, multiplication of the costs by arbitrary lookup constants a_j, of the form:

$$V(i,j) = \max(a_j V(i,j-1) + x_j, y_j) \tag{6}$$

whose applications extend to many dynamic programming problems in several areas. This is concisely described as a *semi-rings* in algebraic dynamic programming.

By partitioning the row into equal sized intervals (e.g., $k = M/4$ in Equation 3 as in Figure 5), we can relate the cost of V_D state at the end of the partition to that at the beginning of the partition. Since the partitions are non-interacting, the symbolic dependencies (i.e., x_j and y_j) can be worked out independently and identically following Equations 4-5. With the help of the above relations, the cost $V_D(i,1)$ is propagated with a stride of $M/4$ to determine the costs of $V_D(i,k)\cdots$ and $V_D(i,(p-1)k)$ in $p-1$ iterations, where p is the number of blocks. Now, since the blocks are completely decoupled from one another, it is possible to fill-in the intermediary values, again independently and identically for each partition. The time required is $2M/p+p$. It is possible to increase the number of partitions so as to achieve larger speed-ups. The space required is $O(p)$ to store and update the variables x_j, y_j for each partition. If the communication latency between the partitions is c, measured in terms of fastest intra-device communication time, then the total time is given by

$$T = 2M/p + pc \tag{7}$$

The implementation needs no additional resources other than parallel updating of the variables x_j and y_j for each partition. The optimal time is obtained by setting, $dT/dp = 1 - 2M/p^2$ to 0. The optimal number of partitions is $\sqrt{2M}$ and the gain in performance due to the data parallelism defined as the ratio of the time for our method over the sequential evaluation time is $M/(2\sqrt{2})$. Though it is possible to implement a tree type parallel-scan reduction at this point, which could be evaluated in $O(log(M))$ steps with $O(M)$ number of processors.

4. RESULTS

For our tests we use the latest NCBI NR protein database [2] which includes 10.54 million sequences and has a size of 5.4GB. The database is periodically updated to include newly discovered protein sequences. For evaluation purposes, we use our implementation to decode this database against three different models of size 128, 256, and 507 respectively. These values were chosen to represent small, medium, and large models respectively. We measure the kernel run time to decode the sequences with 1 and 4 GPU-Tesla C1060s. The entire decoding is carried out on GPUs and therefore the host processor does not affect the performance measurements. The runs were performed several times and the run times were consistent across the different runs.

In our algorithm, the optimal number of partitions (i.e, the way we partition the row as shown in Figure 5) depends on the size of the row that is being reduced, which is equal to the model size. The optimal number of partitions is chosen to be 23 for a model size of 507, 18 for a model size of 256 and 12 for a model size of 128. Contrary to work in [11], each GPU thread block operates on individual sequences and writes the cost of decoding to the global memory independently. Thus there is no need to sort the sequences and it is possible to process an unsorted database without any loss in performance. This is particularly advantageous considering that sequence databases can contain millions of sequences and their sorting can take several hours. Since databases are continually updated, the sorting operation can quickly become intractable, when the size reaches Tera- or Petabytes of data in size. As pointed out in [11], GPU-HMMER suffers from a significant loss in performance when processing an unsorted database. The performance measurements are shown in Figure 6. Figure 6 shows the times our parallel code needs to decode the entire database against models of size 128, 256, and 507 chosen from the *pfam* HMM database when using 1 and 4 GPU-Tesla C1060s. We observe that the time grows linearly with the model size and scales linearly with the number of GPUs. The sequences were decoded with a sensitivity of 100%. The figure also compares our performance versus GPU-HMMER with 4 GPU-Tesla C1060s. Our implementation shows a speed-up of 5X-8X compared to GPU-HMMER. GPU-HMMER was compiled with suggested optimizations from the developers. Our 4-GPU implementation also achieves 100+X speedup compared to a serial implementation on an AMD Opteron at 2.33GHZ [11].

5. CONCLUSIONS

In this paper we describe an approach to extract data parallelism in applications with serializing data dependencies. These dependencies are ubiquitous in a variety of bioinformatics applications. In particular, we apply our method to the *hmmersearch* problem to accelerate the protein-motif finding in a database of 5.4GB. We observe a speedup of 100+X compared to the sequential CPU execution and a speed up of 5X-8X compared to GPU-HMMER using unsorted data. The use of unsorted data ensures versatility and dynamicity of our method across different databases independently from their size and updating frequency.

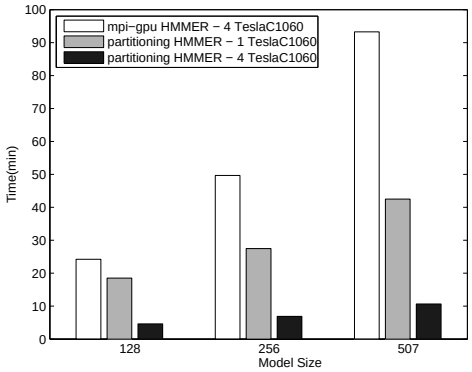

Figure 6: Time comparison to decode the NCBI/NR unsorted protein database of 10.54 million sequences.

6. REFERENCES

[1] Compute Unified Device Architecture. `http://www.nvidia.com/cuda`.

[2] NCBI NR Protein Database. `ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz`.

[3] Eddy, S. Profile hidden Markov models. *Bioinformatics 14* (1998), 755–863.

[4] Eddy, S. HMMER: Profile HMMs for protein sequence analysis. `http://hmmer.janelia.org`, 2004.

[5] Horn, D., Houston, M., and Hanrahan, P. ClawHMMER: A streaming HMMer-search implementation. In *Proc. of ACM/IEEE Supercomputing Conf.* (2005).

[6] Krogh, A., et al. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology 235* (1994), 1501–1531.

[7] Lindahl, E. Altivec HMMer, version 2.3.2. `http://powerdev.osuosl.org/project/hmmerAltivecGen2mod/`.

[8] 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* (2006).

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

[10] Takagi, T., and Maruyama, T. Accelerating HMMER search using FPGA. In *IEICE Tech. Rep., RECONF2009-6* (2009), vol. 109, pp. 31–36.

[11] 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 IPDPS* (2009).