GPU Accelerated Smith-Waterman

Yang Liu[1], Wayne Huang[1,2], John Johnson[1], and Sheila Vaidya[1]

[1] Lawrence Livermore National Laboratory
[2] DOE Joint Genome Institute, UCRL-CONF-218814
{liu24, whuang, jjohnson, vaidya1}@llnl.gov

Abstract. We present a novel hardware implementation of the double affine Smith-Waterman (DASW) algorithm, which uses dynamic programming to compare and align genomic sequences such as DNA and proteins. We implement DASW on a commodity graphics card, taking advantage of the general purpose programmability of the graphics processing unit to leverage its cheap parallel processing power. The results demonstrate that our system's performance is competitive with current optimized software packages.

1 Introduction

Sequence comparison [1] is a fundamental tool for genome scientists to infer biological relationships from large databases of related DNA and proteins sequences. This task cannot be adequately solved by traditional string matching methods because genomic sequences that share the same biological purpose mutate over time when exposed to evolutionary events, and may no longer match identically. Genomic sequence comparison tools such as BLAST and HMMer are based upon approximate string matching principles that measure the overall similarity between strings and are thus more tolerant of mismatches.

This fuzzy string matching problem can be formulated in two different ways. The similarity between two strings can be assessed explicitly, by minimizing an ad hoc cost function (e.g. edit distance) over all possible alignments between the strings. Alternatively, a similarity score can also be computed stochastically, by finding the maximum likelihood path through a hidden Markov model (HMM) trained from an input string. Both of these approaches are optimization problems that require dynamic programming (DP) to solve. The DP step is often very computationally expensive, especially when comparing large strings. Fortunately, the data dependencies in the recurrence relations allow some degree of parallelism and the computation for some cell entries of the DP table can be distributed across a set of processors. This paper describes a parallel hardware implementation of the double affine Smith-Waterman (DASW) alignment algorithm [2][3]. DASW uses DP to find the best local alignment between two genomic sequences by optimizing a scoring function across all possible alignment arrangements, taking into consideration mismatches and gaps in either sequence to maximize the total amount of base pairings.

We chose to implement DASW on a graphics processing unit (GPU) because GPUs are cheaper and better suited for SIMD computation than conventional CPUs and more commercially available than many other special-purpose processors (i.e. ClearSpeed [4]). In fact, graphics cards can already be commonly found in many desktop and laptop computers. Moreover, we can also leverage existing visualization clusters to accelerate genomic sequence comparison between large databases, which often require days to compute on a single processor.

The NVIDIA GeForce 7800 GTX card in our system contains 24 processors with an aggregated peak compute performance of 313 GFLOPS. Unfortunately, the GPU's internal memory bandwidth of 38.4 GB/s limits the performance of most memory-bound applications (such as DASW) to around 70 GFLOPS. However, this is still very impressive, especially given that the estimated retail cost of the graphics card is only about $500. A dual-core Intel Pentium D 840 processor running at 3.2 GHz achieves roughly 25.6 GFLOPS using the SSE3 extensions and costs around $600. In comparison to CPUs, GPUs have much better cost-performance. However, GPUs are also much more difficult to program since they are designed to accelerate computer games and graphics applications. In practice it is difficult for general computational (i.e. non-graphics) tasks such as DASW to efficiently exploit data parallelism on the GPU architecture due to its strict resource constraints, limited data formats, communication overheads, and restrictive programming models. Furthermore, the GPU cannot efficiently share data with the CPU since memory transactions across the PCI-Express bus are very slow (4 GB/s) relative to the processing power and internal bandwidth of the graphics card. Intermediate data must reside completely in texture memory and be processed entirely on the GPU to avoid the bandwidth bottleneck. However, a typical commodity graphics card has only 256 MB of texture memory, which may be insufficient for applications on large data sets. Nevertheless, despite these practical challenges, many computational problems have been already implemented on the GPU, achieving on average, several times speedup over respective optimized software implementations. These performance gains are encouraging for cheap high performance computing and show the potential of GPUs as versatile math co-processors.

2 Related Work

Since genomic sequence comparisons are very expensive to compute in software, several hardware systems have been proposed to accelerate this task. ClawHMMer [5] is a streaming implementation of hmmsearch on a GPU, reporting performance at least twice as fast as the best optimized software packages. ClawHMMer implements the Viterbi algorithm, which uses DP to find the most likely path through a trained HMM network. In order to maximize throughput, ClawHMMer processes several sequence comparisons in parallel. Our system, in contrast, achieves parallelism but processes sequence comparisons one at a time to also minimize the latency of individual comparisons. TimeLogic's DeCypher card uses field programmable gate array (FPGA) chips to implement the logic for DP used by the DASW and HMMer algorithms. FPGAs are favored for their reconfigurability and fast integer arithmetic, but are also nontrivial to program. Rognes, et al [6] reported a six-fold speedup in their Smith-Waterman implementation using Intels MMX and SSE3 extensions by hand-tuning inline assembly instructions. Unfortunately, a notable drawback is that their implementation store alignment scores in 8-bit, and cannot perform long sequence comparisons where the scores are expected to exceed 255.

3 Smith-Waterman Algorithm

The Smith-Waterman algorithm computes the optimal local alignment for a pair of sequences according to a scoring system defined by a substitution matrix and gap penalty function. The substitution matrix is a symmetric matrix that assigns the cost of pairing bases together. The costs are derived from the observed substitution frequencies in alignments of related sequences. Each potential base pair is given a score representing the observed frequencies of such an occurrence in alignments of evolutionarily related sequences. This score also reflects the sample frequency of each base since some bases occur more frequently in nature than others. Identities are usually assigned the highest positive scores, frequently observed substitutions also receive positive scores, but matches that are observed to be highly unlikely are penalized by negative scores. The two most popular sets of substitution matrices for comparing long sequence are the BLOSUM and PAM matrices [7][8]. Smith-Waterman also supports gaps in the sequences at a penalty to maximize the substitution score. The parameters of the gap penalty function influence the length and frequency of gaps allowed in the alignment. There are generally three types of gap penalty functions:

$$\begin{aligned} \textbf{constant} &: g(n) = bn \\ \textbf{single affine} &: g(n) = a + bn \\ \textbf{double affine} &: g(n) = a + min(n,k)b_0 \\ & \qquad\qquad +max(0, n-k)b_1 \end{aligned} \tag{1}$$

The constant gap function assigns a fixed cost to each gap space, regardless of its placement in the alignment. The single affine gap function penalizes gap creation to encourage the placement of new gap spaces to extend existing gaps rather than opening new ones. This is a more plausible model for gaps in genomic sequences since a gap of more than one space can be accounted for by a single evolutionary event. The double affine gap function extends this idea by assessing a separate penalty for each gap space that extends a gap beyond the threshold of spaces; is usually set smaller than to encourage longer gaps. We implement this function in our system since it generalizes both the constant and single affine gap functions. The optimal sequence alignment according to this scoring system is found by evaluating a set of recurrence relations over each cell of the DP table. For example, the following relations compute the optimal alignment using the single affine gap penalty function:

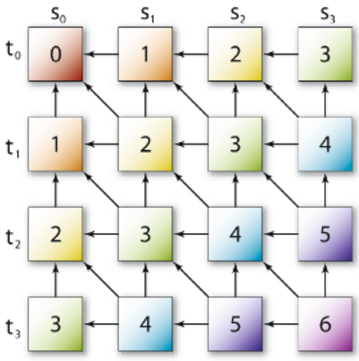

Fig. 1. Data dependency for DP computation. Cells from each diagonal are mutually independent, and depend only on cells from the previous two diagonals.

$$
\begin{aligned}
E_{0,j} &= E_{i,0} = D_{0,j} = D_{i,0} = I_{0,j} = I_{i,0} = 0 \\
E_{i,j} &= max\{0, E_{i-1,j-1} + match(s_i, t_j), D_{i,j}, I_{i,j}\} \\
D_{i,j} &= max\{E_{i-1,j} + a, D_{i-1,j} + b\} \\
I_{i,j} &= max\{E_{i,j-1} + a, I_{i,j-1} + b\} \\
M_{i,j} &= max\{E_{i,j}, E_{i-1,j}, E_{i,j-1}\}
\end{aligned}
\tag{2}
$$

$\mathbf{s} = s_0 s_1 \ldots s_{m-1}$ and $\mathbf{t} = t_0 t_1 \ldots t_{n-1}$ are the two input sequences, $match$ is the substitution cost matrix, and a, b are the single affine penalties for respectively opening and extending a gap. Implementing the double affine gap penalty function requires the intermediate gap lengths to be associated with $D_{i,j}$ and $I_{i,j}$ to select the appropriate penalty for b. The purpose of $M_{i,j}$ is to track the maximum alignment score for all cells (intermediate alignments) in the DP table – the final maximum alignment score will be propagated to the last cell $M_{m-1,n-1}$. Pointers for each $M_{i,j}$ are also maintained to track the cell location with the maximum alignment score and simplify the alignment trace back step.

Figure 1 illustrates the data dependencies involved in computing the recurrence relations over the DP table. In a single processor system, DP cells are processed sequentially, but a multiprocessor system can efficiently exploit the data dependencies by processing independent cells from each DP table diagonal (up to $min(m, n)$ cells) in parallel. Furthermore, the data dependencies also allow opportunities for cache optimization; only two diagonals are accessed during a computation pass so a sufficiently large LRU cache can maximize its cache coherency. With $p = min(m, n)$ processors, the DP table can be computed in $(m + n - 1)$ passes by sequentially processing each diagonal. Unfortunately, there is some efficiency loss since a few processors must stall when processing non-major diagonal. The total number of stalls in the DP computation is $p(p-1)$. However, the query sequence is commonly matched against a database of many target sequences, and the computation of the DP tables can be interleaved together to amortize the processor stalls.

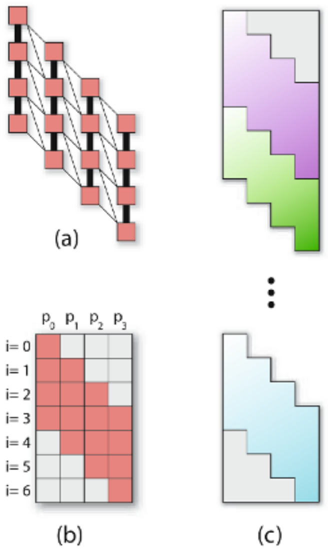

Fig. 2. The data dependency shown in (a) allows cell computations to be assigned to a set of processors as shown in (b). The wasted space can be amortized over several query sequence comparisons by connecting their DP tables together as shown in (c).

4 GPU Implementation

We implement DASW on the NVIDIA GeForce 7800 GTX graphics card using the OpenGL API, and the GL shading language (GLSL). Our implementation only involves two stages from the OpenGL rendering pipeline: geometry transformation and fragment rasterization. The geometry serves as the proxy that initializes the pipeline for the DP computation and defines the area of computation. After copying the query and target sequence data to texture memory, for each diagonal, the geometry transformation stage is passed (the vertices of) a quadrilateral that can compactly contain the DP cells of the diagonal. The dimensions of this quadrilateral must be carefully chosen to minimize wasted cells and to take advantage of any tiling optimizations on the GPU. The geometry transformation stage assigns to each constituent fragment from the quadrilateral a unique texture coordinate address and then engages the fragment rasterization stage, where the DP recurrence relations are evaluated over the fragments by a set of processors. The resulting pixel values are stored into an image buffer in texture memory, to be reused in subsequent passes. This computation loop proceeds until all diagonals have been processed. The last cell in the DP table contains the optimal alignment score $M_{m-1,n-1}$ and is retrieved from texture memory. If alignment generation is desired, then up to $(m + n - 1)$ more pixels are copied out of texture memory, comparing the intermediate values of $E_{i,j}$ with $D_{i,j}$ to build the alignment.

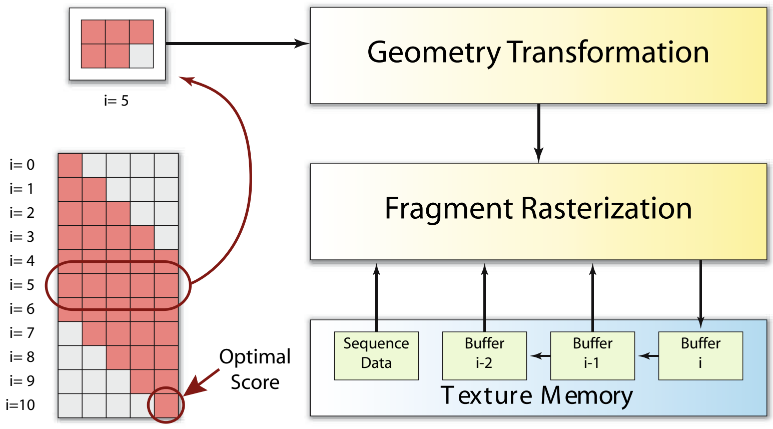

Fig. 3. OpenGL rendering pipeline. An execution pass is initiated at the geometry transformation stage by drawing a quadrilateral representing the cells of a DP table diagonal. The cells are processed by the fragment rasterization stage and the results are saved to a queue of buffers in the texture memory, to be accessed for subsequent passes. Computation proceeds until the last cell is processed; this cell contains the optimal alignment score.

In each execution pass, a fragment processor can output a maximum of 16 components per pixel. Although each component can be represented in 32-bit floating point, we opted to represent our data using 16-bit floats to save storage and bandwidth. In our DP table, each cell must maintain $E_{i,j}$, $D_{i,j}$, $I_{i,j}$, two gap lengths, and the local maximum alignment score along with its respective pointer (represented by two components), requiring a total of 8 components. Since there are 16 components available, we can compute and store the data of two DP table cells per pass in each pixel. We batch the data such that two DP tables are computed simultaneously to amortize the cost of initiating each execution pass through the pipeline. The target sequences are batched as follows: we divide the set of target sequences into two sets of sequences with roughly the same total number of bases and form an aggregate target sequence from each set by concatenating individual target sequences together, separating them by a special delimitation character. The two aggregate sequences, along with the query sequence are then copied into texture memory. When the fragment processor encounters the special delimitation character during a cell computation, it sets its corresponding pixel component values to zero. This introduces a little extra overhead cost but is convenient since it effectively resets the initial conditions for the next DP table computation, which allows us to interleave query-target comparisons as shown in Figure 2.

Since only high scoring sequence alignments are interesting candidates for alignment trace back, actual score, we implemented two modes of DASW on the GPU: one that supports alignment trace back (ATM), and another faster version that only computes the alignment score (ASM). In order to compute the alignment trace back, ATM requires the entire DP table to fit within texture

Table 1. Table of results for a single query sequence (16,384 bases) compared against 983 target sequences (462,862 bases)

Platform	Total time (sec)	Throughput (10^6 cells/sec)
CPU (*osearch34*)	147.46	51.43
CPU (*ssearch34*)	63.17	120.05
GPU (ATM)	42.51	178.41
GPU (ASM)	31.45	241.12

memory (a graphics card with 256 MB of texture memory can store roughly 2^{22} DP cells). Otherwise, if the texture memory overflows, the resulting paging across the PCI-Express bus will cripple the GPU's performance. In contrast, ASM only needs to store three diagonals worth of DP cells in texture memory. Furthermore, ASM does not need to maintain any pointers, so each pixel only needs to store 12 components (two DP cells), which amounts to roughly a 25% savings in total bandwidth over ATM. ASM is faster than ATM and can be used to filter out poor matches. If high homology is not expected, ASM can be used to identify high scoring query-target comparisons in a first pass, so that they can be recomputed by ATM to generate their full alignments in a second pass.

5 Results

We benchmark our system using two reference programs selected from the FastA suite [9]: *osearch34* is a software implementation of Smith-Waterman using single-affine gap penalties that takes advantage of several caching optimizations. *ssearch34* extends this implementation by also including Phil Green's SWAT optimization [10]. SWAT follows a heuristic that ignores paths through the DP table where the score would be less than the gap open penalty and essentially allows the algorithm to skip the computation for some cells. However, the performance of SWAT highly depends on the gap penalty value; it is not very useful for small gap penalties and cannot be used at all if very high gap penalties are required. Our test system is a 3.2 GHz Pentium D 840 processor with 2 GB of RAM, equipped with a NVIDIA GeForce 7800 GTX card. We measured the performance of our system by aligning a single query protein sequence consisting of 16,384 amino acids against a database of 983 protein sequences, altogether consisting of 462,862 amino acids, a computation of roughly 7.5 billion DP cells, which is representative of problem sizes in genomic sequence comparison. We used BLOSUM62 for the substitution cost matrix with single affine gap penalties $a = -12$, $b = b_0 = b_1 = -2$, and $k = 0$. These parameters are typical for most genomic sequence comparisons.

6 Discussion

The Smith-Waterman algorithm is a computationally-intensive string matching operation that is fundamental to the analysis of proteins and genes. Our novel implementation of the Smith-Waterman algorithm exploits the parallel processing power of GPUs to achieve a two-fold speedup over the best optimized software implementation. Our system is general enough to support arbitrarily complex gap penalty functions and allows very long sequence comparisons (query sequence sizes up to 2^{22} bases and target sequences of unlimited length). The memory bottleneck of our system limits its computational potential. This situation may improve as GPUs increase their internal bandwidth, expose caches to reduce communication costs, or allow instruction scheduling to conceal memory latency. Our future work includes building a threaded cluster implementation to distribute the work for genomic sequence comparisons among several CPU and GPU nodes to take advantage of existing visualization clusters equipped with high-end GPUs. Furthermore, our GPU framework for evaluating DP extends to other optimization problems, and we are interested in identifying applications that can benefit from this acceleration.

References

1. Brenner, S., Chothia, C., T.J.P., H.: Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. Proc. National Academy of Science **95** (1998) 6073–6078
2. Gotoh, O.: An improved algorithm for matching biological sequences. Journal of Molecular Biology **162** (1982) 705–708
3. Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology **147** (1981) 195–197
4. ClearSpeed: Advance^{TM} Board, http://www.clearspeed.com/index.html. (2006)
5. Horn, R., Houston, M., Hanrahan, P.: ClawHMMer: A streaming HMMer-search implementation. Proc. Supercomputing (2005)
6. Rognes, T., Seeberg, E.: Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics **16** (2000) 699–706
7. Henikoff, S., Henikoff, J.: Amino acid substitution matrices from protein blocks. Proc. National Academy of Science **89** (1992) 10915–10919
8. Pearson, W.: Effective protein sequence comparison. Meth. Enzymol **266** (1996) 227–258
9. Pearson, W., Lipman, D.: Improved tools for biological sequence comparison. Proc. National Academy of Science **85** (1988) 2444–2448
10. Green, P.: SWAT Optimization, http://www.phrap.org/phredphrap/general.html. (2006)