Fast Exact String Matching on the GPU

Michael C. Schatz and Cole Trapnell

Abstract— We present a string-matching program that runs on the GPU. Our program, Cmatch, achieves a speedup of as much as 35x on a recent GPU over the equivalent CPU-bound version. String matching has a long history in computational biology with roots in finding similar proteins and gene sequences in a database of known sequences. The explosion in sequence data available in the 80s and 90s motivated the development of ever faster techniques for searching for similar sequences, and ultimately lead the use of parallelized execution of string matching algorithms using sophisticated data structures called suffix trees. Suffix trees can be constructed time proportional to the length of the corpus, and provide exact matching of a query in time proporional to the length of the query, independent of the size of the corpus. Here, we present our string-matching kernel for use in the Compute Unified Device Architecture, which executes parallelized searching of a suffix tree for finding exact matches for a set of query strings. We compare our GPGPU suffix tree search to a serial CPU version of the algorithm, and analogous components of the widely used CPU program MUMmer, and explore issues associated with storing a suffix tree in a graphics card's memory, and data distribution among the GPU's processing units.

Index Terms—computational biology, GPGPU, suffix tree, string matching, data reordering.

1 Introduction

Genetic information is passed from parent to offspring in the biological macromolecule DNA, and is encoded in the individual's sequence of 4 different nucleotides. The full sequence of nucleotides for an organism, its genome, varies in length from hundreds of thousands of nucleotides for genetically simple organisms, such as bacteria, to billions of nucleotides for genetically complex organisms, such as humans. Genes are the regions of the genome that encode for proteins, which are the functionally active molecules in an organism. Gene sequences specify a protein's sequence of component molecules called amino acids. Each triple of nucleotides in the DNA of a gene specifies one of the 20 possible standard amino acids. The chain of hundreds or thousands of amino acids specified by the gene folds into a very complex configuration, and the physical shape of a folded protein determines its biological function.

Pairs of similar sequences are biologically related under the commonly held assumption that two homologous (highly similar) protein sequences will fold into similar configurations, and thus have a similar biological function. Given a database of sequences with known function and a novel sequence with unknown function, one can use a sequence alignment algorithm to discover homologous sequences, and thereby infer the function of the novel sequence. A sequence alignment algorithm reports the biological similarity between a pair of sequences, by modelling potential mutations between the sequences.

Early work on detecting homologous sequences focused on creating optimal algorithms that would exactly compute and report the similarity score between pairs of sequences. However, improvements in high throughput sequencing technology led to an explosion in number of known sequences, and motivated research for more efficient methods for detecting similar sequences. This inspired the development of heuristics that find highly similar alignments very quickly at the cost of potentially not reporting lower similarity sequences.

Furthermore, starting with the second sequenced species of Mycoplasma in 1996 (the third [1] and fifth [2] completely sequenced prokaryotes), computational biologists entered into the era of comparative genomics, in which the entire genomes of organisms are compared. In comparative genomics, the relatively simple techniques used for scanning a single short query sequence against a database of known sequences proved too inefficient yet again, and required the use of even faster algorithms, using sophisticated data structures such as suffix trees.

Even with the highly efficient algorithms and data structures invented, modern computational biologist still rely on computational grids consisting of many computers executing algorithms in parallel to increase the throughput of their searches. Our work acts to replace an entire computational grid of computers used for sequence alignment with a single highly parallel commodity multiprocessing board, in the form of a high performance Graphics Processing Unit (GPU) programmed in the Compute Unified Device Architecture (CUDA) framework.

2 Related Work

2.1 Sequence Alignment

Sequence alignment has a long history in computational biology. The earliest algorithms, such as the Needleman-Wunsch [3] global alignment and Smith-Waterman [4] local alignment algorithms from the 70s and early 80s, were developed to detect similarity between a pair of DNA or protein sequences. These early algorithms use dynamic programming to compute optimal alignments between a pair of sequences in time proportional to the product of their lengths.

In the 80s and 90s, the number of DNA and protein sequences grew exponentially and it quickly became infeasible to compute an optimal alignment between a query sequence and every sequence in the database of known sequences. As a result, researchers developed heuristic techniques such as FASTA [5] and BLAST [6] to more quickly discover highly similar alignments, while not reporting more distant alignments. Researchers found this to be an acceptable trade-off between sensitivity and speed, since in most applications only the highly similar alignments are relevant. Consequently, BLAST became the de facto bioinformatics tool of choice for finding homologous gene or protein sequences.

These techniques use the key insight that two highly similar sequences must share common substrings, although possibly separated by relatively short sequences of mutated residues. They use this insight by pre-processing the database to construct a catalog of the short fixed-length substrings found in the database sequences. After this pre-processing, each query string is processed by first finding exact matches between substrings of the query string and the catalog of database substrings. Using a hash table or similar technique, this executes in time proportional to the number of occurrence of each short substring in the database. These short exact matches then act to seed longer, possibly inexact alignments, by connecting together overlapping exact matches, and using Smith-Waterman local alignments to align regions with mismatching characters. The time to pre-process the database may be considerable, but is amortized by constructing it once for many query

- *Michael C. Schatz is with the Center for Bioinformatics and Computational Biology, University of Maryland, E-Mail: mschatz@umd.edu.*
- *Cole Trapnell is with the Department of Computer Science, University of Maryland, E-Mail:cole@cs.umd.edu*

searches, after which the seed-and-extend style algorithms run in time proportional to the length of the query string and relatively independent of the number of strings in the database.

Unlike protein searches, comparative genomics commonly performs comparisons between the much longer whole genome sequences of related organisms. As such, the two genomes will typically share many very long substrings. Consequently, BLAST and related tools which seed their alignments using short fixed length substrings will process many seeds from these exact matches, and thus run relatively inefficiently. This motivated research in the late 90s for new alignment algorithms which could be used for aligning sequences of any length and potentially containing very long exact matches.

One of the most successful algorithms developed for this purpose was a seed-and-extend algorithm called MUMmer [7-9], which uses variable length maximal exact matches to seed in-exact alignments and thus overcome those limitations of BLAST. A maximal exact match is an exact match than cannot be extended at either the beginning or end of the sequences without introducing a mismatch. Therefore, rather than finding many overlapping fixed length exact matches in a BLAST-like system, MUMmer will consider only maximal matches without having to explicitly combine overlapping shorter matches. This is accomplished by performing a more sophisticated pre-processing of the database, which constructs a suffix tree of their sequences.

The suffix tree [10] for string S is a tree whose edges are labelled with substrings of S, and encodes every suffix of S with a unique path from the root to a leaf. There are n leaf nodes for each of the n suffixes with a special terminal character (normally $) added to the end of the string to guarantee a unique leaf node for each suffix. Every internal node has at least 2 children, and represents positions where repeated suffixes diverge. Note a suffix tree is also an efficient representation for all substrings of S because every substring of S from i to j, is the prefix of the suffix starting at i, and thus falls on the path from root to leaf i. A database of multiple strings can be encoded into a generalized suffix tree by marking leaf nodes with the string in which they originate. A suffix tree can be naively constructed in $O(n^2)$ time and O(n) space by iteratively inserting all n suffixes, but can be constructing in O(n) time and O(n) space for a string over a fixed alphabet, such as for DNA or amino acids, by more carefully exploiting the relationships between suffixes. The more sophisticated algorithms make extensive use of additional pointers called suffix links which connect nodes along the path to the ith suffix to same relative position on the $i+1^{th}$ suffix.

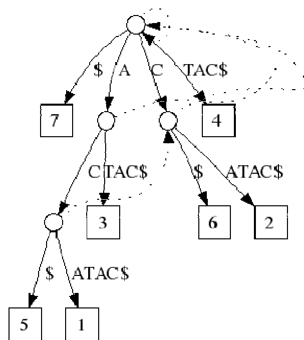

Figure 1. Suffix tree for the string ACATAC$. Leaf nodes are displayed as rectangles labeled with the starting position for that suffix, internal nodes as circles, and suffix links as dashed lines.

Given a generalized suffix tree T from a database of sequences, and a query sequence Q, the presence of Q in the database is determined by navigating the tree from the root following the characters in Q. If at any point, it is not possible to navigate in the tree to the next character in Q, then Q is not present in the database. This is accomplished in time proportional to the length of Q completely independent of the size of T. The algorithm for finding substrings of Q that exactly match T uses similar tree navigation but utilizes the suffix links whenever a mismatch is encounter. The algorithm begins by walking the tree from root using the first characters of Q. When a mismatch occurs at position i such that character Qi is not present in T, then the algorithm reports [1,i-1] as an exact match. The algorithm then follows the suffix link and resumes navigating down the tree starting with character Qi until reaching the next mismatching character Qj (j >= i), and reports [2,j-1] as the next match. If j > i, then Q[1,j-1] cannot be in T or it would have been already been discovered. The key to this algorithm is the use of suffix links which allows the algorithm to continue processing the successive character of Q without having to ever reprocess previous characters.

In an extreme form of comparative genomics, researchers at the Center for Bioinformatics and Computational Biology at the University of Maryland utilize MUMmer and their 128 node computational grid to perform whole genome alignments between all pairs of known genomes (currently 3684 genomes) in their Insignia system. [11] Insignia combines these alignments to generate "signatures" for a set of genomes, which are DNA sequences of user specified length common to the selected set of genomes but are not present in any of the other known genomes. This technology has been used to compute DNA signatures for various pathogens enabling laboratory protocols for highly their efficient and accurate detection.

2.2 GPGPU Programming

As the GPU has become increasingly more powerful and ubiquitous, researchers have begun exploring ways to tap its power for non-graphics, or *general-purpose* (GPGPU) applications. [12] This has proven challenging for a variety of reasons. Until recently, GPUs included two distinct classes of specialized graphics processors: vertex processors, which compute geometric transformations on meshes, and fragment processors which shade and illuminate the rasterized products of the vertex processors. Recently, though, GPUs have relaxed the requirements for using these processors, and have enabled GPGPU programming with non-graphical output. Modern GPUs include several (tens to hundreds) of each type, so both traditional and GPGPU applications are faced with parallelization challenges. [13]

While GPGPU applications have found success in many disciplines, most GPGPU successes stem from scientific computing or other areas with a strong numerical computational component. [14, 15] This includes several ports of bioinformatics applications to graphics hardware. Liu et al implemented the Smith-Waterman algorithm to run on the nVidia GeForce 6800 GTO and GeForce 7800 GTX, and reported an approximate 16x speedup. [16] Charalambous et al ported an expensive loop from RAxML, an application for phylogenetic tree construction. On the nVidia GeForce 5700 LE, the authors achieved 1.2x speedup. [17]

The success of scientific and numeric GPGPU applications is primarily because graphics hardware is organized around a streaming, data-parallel model. On-chip caches for the processing units on GPUs are typically small (often limited to what is needed for texture filtering operations) compared to general purpose processors, which feature caches measured in megabytes. Thus, reads and writes to on-board RAM can have high latency relative to these operations when performed by a CPU on main memory. Furthermore, until recently, vertex and fragment shader programs did not support unrestricted writes by any processor to the on-board RAM, called *scatter* operations. This restriction made implementing certain types of data-parallel applications, such as finite element solvers, more difficult because these applications require frequent exchange of intermediate results among parallel processors. In general, the applications that have performed well in a GPGPU model are those that can decompose their problems into highly independent components and have a high *arithmetic intensity* – those

with an instruction mix that favours register-only operations greatly over operations that address on-board RAM. [18]

nVidia's new G80 architecture radically departs from the traditional vertex+fragment processor pipeline. The G80 features a single unified set of processors that can function as either vertex or fragment processors depending on the needs of an application. Furthermore, nVidia anticipated the benefit such a unified architecture for GPGPU computing, and released the Compute Unified Device Architecture (CUDA) SDK to assist developers in creating non-graphics applications that run on the G80 and future GPUs. CUDA offers improved flexibility over previous GPGPU programming tools, and does not require the application writers recast the operations in terms of geometric primitives as was required by earlier GPGPU environments. In addition, CUDA supports the generic CPU gather and scatter operations for on-board memory. [19]

This improved flexibility does not solve the more fundamental problems caused by the stream-computing organization: the small cache size and associated high latency. However, the G80 architecture also includes (2D) cached memory nominally used for caching texture map lookups during rasterization that can be exploited by GPGPU programs. Several software techniques exist to maximize the benefit offered by a cache. One such class of techniques involves reordering either the data in memory or the operations on those data in order to maximize data and temporal locality. Mellor-Crummey et al reported significant speedup in particle interaction simulations, which feature highly irregular access patterns, by reordering both the locations of particles in memory and the order in which interactions were processed. They tested a re-ordering strategy based on space-filling curves, such as the Hilbert and Morton curves. [20] Another less exotic technique, but one with more relevance to our application, was proposed by Bender et al. The authors give a greedy algorithm for laying out nodes of a tree (or trie) in a blocked memory hierarchy so as to minimize the number of block transfers. [21]

3 Methods

The Cmatch algorithm performs highly parallelized exact string matching on the GPU. Each query sequence is matched against the reference sequence in time proportional to its length by navigating the suffix tree of the reference on the GPU from the root towards a leaf. If the query is present in the reference sequence one or more times, then the algorithm reports the node in the suffix tree which contains the last character of the query. From this, the algorithm can report the number of occurrences and positions of the query in the reference in time proportional to the number of occurrences of the query in the reference.

The suffix tree of the reference sequence is constructed using existing methods [22] in linear time on the CPU. The tree is then "flattened" into 2 2D textures, the node texture and the children texture. The information of each tree node is stored in a pair of 16 byte texels (texture elements). Half of the information for a node, including the start and end coordinates of the edge string in the reference, and the parent pointer, are stored in the node texture at location (i,j). The remaining information for a node, the pointers to the A,C,G & T children, are stored at the same location in the children texture.

In the CUDA architecture, a program can utilize textures for storing large read-only data, and reads from textures are cached using a proprietary 2D caching scheme, optimized for applying textures for graphics applications. Therefore, the algorithm attempts to optimize the 2D locality of the tree structure in these textures by organizing the nodes in 32x32 texel blocks. Near the root of the tree (node depth < 16), the nodes are assigned using a level-order (breadth-first) traversal of the tree which ensures all of the nodes near the root of the tree are placed in the first 32x32 texel blocks. This layout guarantees that all of the children of a given node will be at (nearly) adjacent cells in the texture, and thus maximizes the cache effectiveness near the root of the tree where all of the query strings will be following similar paths. Thus, loading a single 32x32 block for one thread is likely to be needed for the other threads running in parallel. Further from the root (depth >=16), the nodes are arranged so that a node, its children, grandchildren, and great-grandchildren are placed in the same 32x32 block. At this level of depth, the different query strings can be highly divergent, and thus the texture cache is better utilized for cache locality of an individual query.

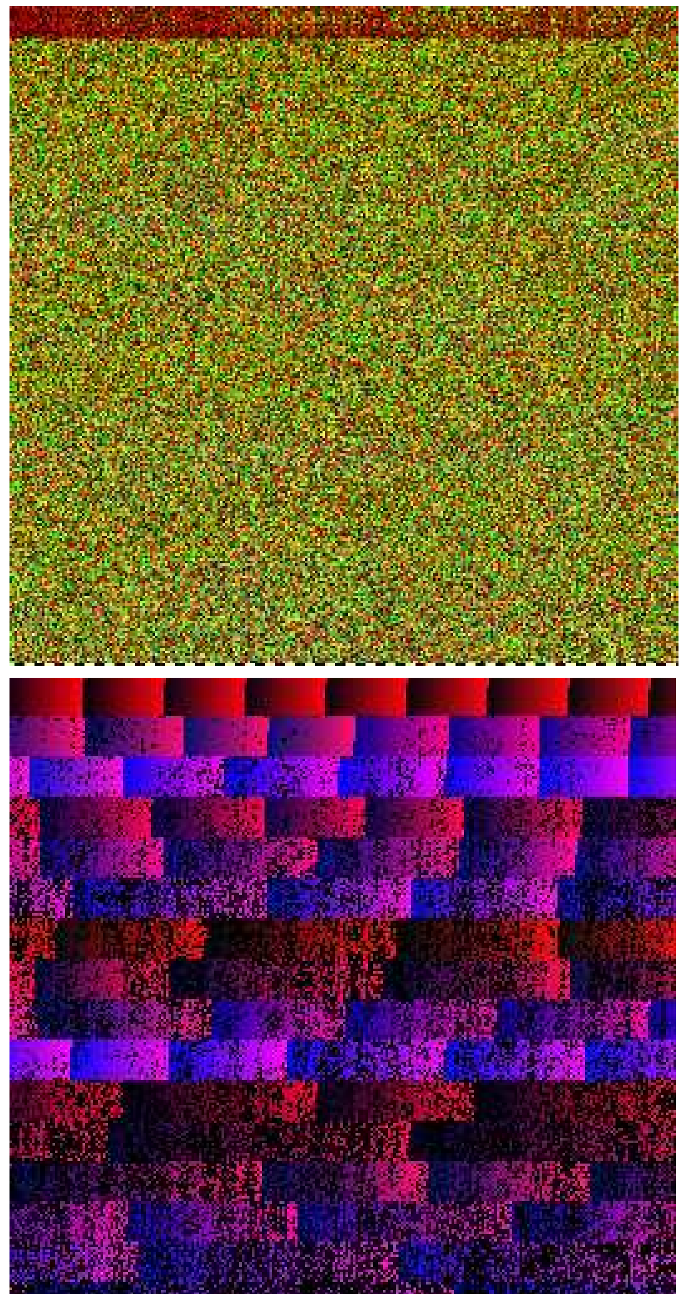

Figure 2 A portion of the suffix tree of the *Bacillus anthracis* genome as stored in two textures: one containing the nodes' parent and '$' links and the start and end coordinates in the corpus for the edge (top), and one for the nodes' child links (bottom)

The set of query sequences are concatenated into a single large buffer (separated by null characters) and transferred in bulk to the GPU. An auxiliary 1D array is created with an entry for each query with the offset into the query buffer for the beginning of that query. The corpus sequence for the tree is also transferred to the GPU as a third texture, so that multiple characters along an edge can be accessed very quickly.

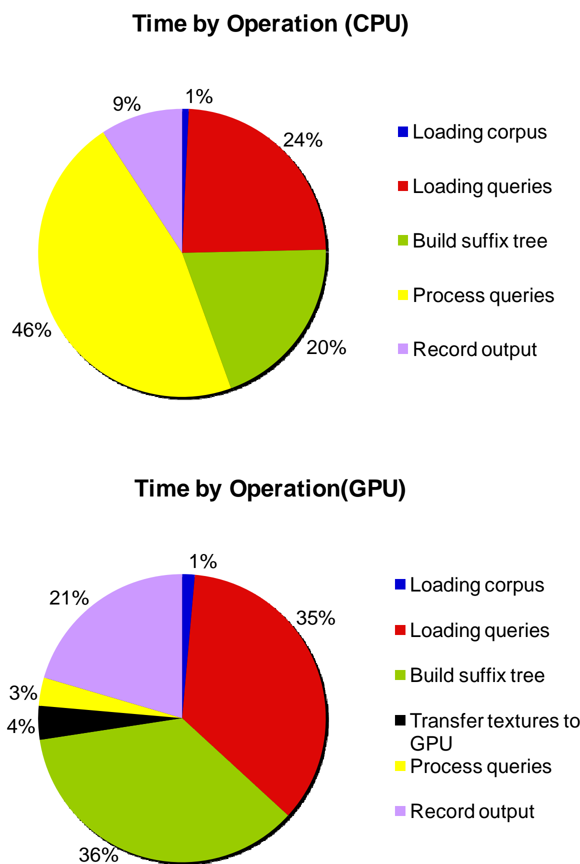

Figure 3 The time spent in each phase of the suffix tree matching program on the CPU (a) and GPU (b)

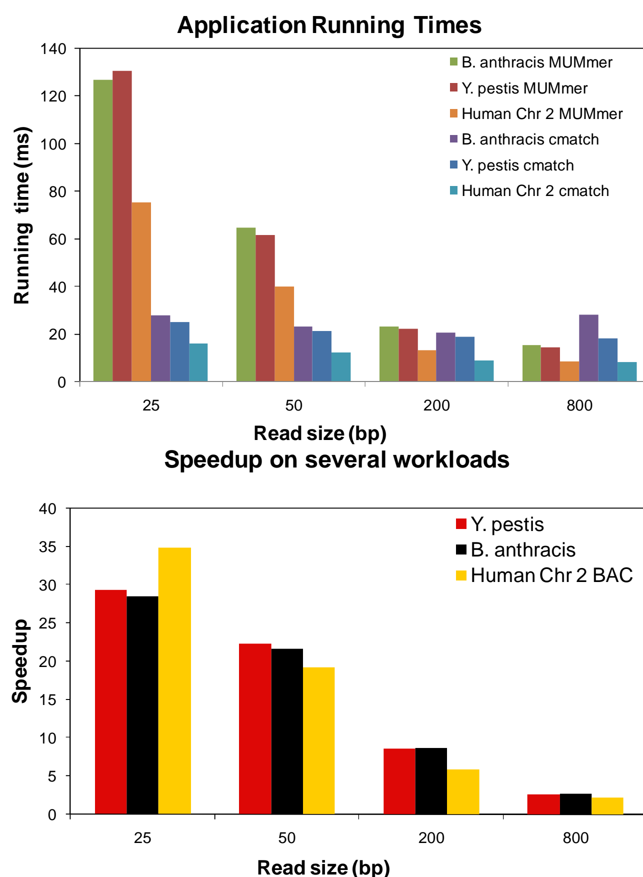

Figure 4 (a) The time to run the each full query set with Cmatch and MUMmer. (b) Speedup achieved by the kernel running on the GPU vs. on the CPU

Each multiprocessor on the GPU is assigned a subset of queries to process in parallel. The executable code running on each processor, the *kernel*, matches its assigned query sequence to the reference by navigating the tree stored as texels according to the sequence of its assigned query. If the entire query matches the reference then the id of the last node visited is stored in an output buffer stored in global memory on the GPU. Special care was taken to ensure the number of registers used by the kernel matching program was minimal to ensure full occupancy on the multiprocessors, and that branching was eliminated wherever possible.

After execution is complete for all of the queries the output buffer is read back by the CPU. The node id in the buffer is sufficient to report if an individual query is present anywhere in the reference. The program uses this node id to optionally report the number of occurrences of the query in the reference and/or the positions of the query in the reference. Both are computed in time proportional to the number of occurrences of the query in the reference, by traversing the tree from the reported node to all of the leaves in that node's subtree.

4 Results and Discussion

We measured the performance of the Cmatch parallel exact string matching algorithm executed on the GPU versus the same algorithm running in serial on the CPU. We also compared the running time for Cmatch against that of MUMmer on the same workloads. The GPU used was an nVidia GTX 8800 with 16 multiprocessors and 768 MB of on board RAM, allowing for up to 128 simultaneous computations at 1.35 GHz. The CPU used was a 3.0 GHz Intel Xeon processor with 2GB of RAM.

The dataset composed of 3 reference sequences: the bacterial genomes of *Yersinia pestis* (4.6 Mbp in length) and *Bacillus anthracis* (5.20 Mbp), and one simulated bacterial artificial chromosome (BAC) of *Homo sapiens*.. The simulated BAC consists of the first 200,000 bp of NCBI build 36 of chromosome 2. For each reference, we generated four sets of queries, each containing a total of 25 Mbp of query sequence. The query sets were constructed of randomly chosen subsequences from each corpus to simulate current sequencing technologies. The sets were as follows: 10 million queries of 25bp in length, 5 million of 50bp, 1.25 million at 200bp, and 312,500 queries at 800bp.

For each combination of the 3 references and 4 query sets, we recorded the wall-clock execution time when executing in parallel on the GPU versus running the same algorithm serially on the CPU. These times include the fixed overhead CPU time to read the reference sequence and construct the suffix tree, and read the query sequences from disk. We also separately measured the time spent executing the match kernel on the GPU and match code on CPU.

Figure 4b shows the speedup (CPU time / GPU time) achieved by our GPU matching kernel over the sequential CPU version for each query set. The speedup for the 25bp query set is as high as 35x, but substantially worse on 800bp reads, at only 2x. We believe the poor performance on longer reads is due to several factors, including poorer cache hit rate during traversal of the suffix tree further from the root, and *thread divergence*, which occurs when two threads running the same kernel and on the same multiprocessor execute different instructions at a certain point in the kernel's execution. This

occurs as the result of branch instructions generated by `if` or `while` statements in the kernel. These instructions are used by the Cmatch kernel to process the sequential characters of the query string and follow the appropriate edges of the tree. When threads diverge, the multiprocessor is forced to serialize their respective instruction streams rather than running the threads in parallel. Further investigation is needed to identify and correct the source of poor performance on long read query sets.

Figure 3b shows the percentage of time spent in the different phases of our GPU-based program while matching 25bp queries against the *B. anthracis* genome. Figure 4a shows the percentage of time spent matching in the different phases of our CPU-based program for the same data set. The time spent matching the queries on the GPU was only 3% of the total application running time compared to nearly 50% of the time when running on the CPU. Unlike the CPU application, the GPU version's time is dominated by reading the queries from the disk, and constructing the suffix tree.

Figure 4a shows the whole-application running times for Cmatch and MUMmer on the workloads detailed above, including time spent in I/O and other non-matching portions of the code. While MUMmer outperforms Cmatch on workloads with a small number of relatively long reads, Cmatch offers a ~5x speedup over MUMmer on workloads with many short queries. Further, Cmatch's running time grows very slowly as the number of reads increases, reflecting the dominance of I/O in running time. MUMmer is a mature application that offers many more features than Cmatch, but for processing many (>1,000,000) short queries, Cmatch is the faster application.

Table 1 shows the texture memory footprint of the suffix tree constructed for each corpus. Bacterial genomes are small relative to the eukaryotic genomes that have been sequenced to date, but even these small genomes yield suffix trees that occupy roughly one third of the available GPU memory on our G80. We plan to address this limitation in future version of Cmatch by investigating other techniques for encoding the suffix tree in memory, and possibly using a suffix array representation for matching.

Corpus	Corpus Length	Suffix tree size (MB)
Human Chr 2 BAC	200000	14.125
B. anthracis	5,227,403	266.75
Y. pestis	4,595,073	234.125

Table 1 Suffix tree size for the test seqeunces.

FUTURE WORK

We plan to improve our GPU matching program by increasing performance and capacity. Unfortunately, details on the texture caching mechanism on the G80 are not readily available. We hope that as details emerge either from nVidia or through empirical investigation we are able to improve the cache hit rate while traversing suffix trees. We believe this will help improve poor performance for queries with 800bp or longer reads. We would also like to thoroughly investigate the prevalence of thread divergence in our matching kernel.

We also plan to extend the matching kernel to support substring matching. Supporting substring matching on the GPU would be of immediate practical value to computational biologists, and would allow for more sophisticated analysis performed, including inexact matching with a seed and extend algorithm such as used by MUMmer. The primary challenge of this extension is a given substring of the query may match at any number of positions on the reference, but these positions must be reported through fixed sized buffers transferred from the GPU to the CPU.

We plan to address the limitation of corpus size by segmenting the corpus and constructing a suffix tree for each segment. The full query set may then be run on each segment suffix tree. Care must be taken for matches that span the segments, but we believe that such a segmentation scheme will relax the corpus limitation and allow our GPU matching program to run queries even on large eukaryotic genomes.

CONCLUSIONS

Operations on the suffix tree have extremely low arithmetic intensity – they consist mostly of following a series of pointers. Thus, string matching with a suffix tree lookup is expected to be a poor candidate for a parallel GPGPU application. However, our results show that a significant speedup, as much as 35x, can still be achieved through the use of cached texture memory and data reordering to improve access locality. This speedup is currently realized only for large sets of short queries, but these read characteristics are beginning to dominate the marketplace for genome sequencing as technologies such as by Solexa improve and create on the order of 200 million 50 bp reads in a single run. Thus our application should perform extremely well on workloads commonly found in the near future. The success of our application is in large part the result of the first truly general purpose GPU environment that is CUDA, which allowed us to formulate and implement our algorithm directly using a tree data structure. We therefore expect essentially any highly parallel algorithm to perform extremely well on a relatively inexpensive GPU, and anticipate widespread use of GPGPU technologies in the near future.

ACKNOWLEDGEMENTS

The authors wish to thank Steven L. Salzberg for funding the GTX 8800.

REFERENCES

[1] C. M. Fraser, J. D. Gocayne, O. White, M. D. Adams, R. A. Clayton, R. D. Fleischmann, C. J. Bult, A. R. Kerlavage, G. Sutton, J. M. Kelley, and et al., "The minimal gene complement of Mycoplasma genitalium," *Science*, vol. 270, pp. 397-403, 1995.

[2] R. Himmelreich, H. Hilbert, H. Plagens, E. Pirkl, B. C. Li, and R. Herrmann, "Complete sequence analysis of the genome of the bacterium Mycoplasma pneumoniae," *Nucleic Acids Res*, vol. 24, pp. 4420-49, 1996.

[3] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J Mol Biol*, vol. 48, pp. 443-53, 1970.

[4] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J Mol Biol*, vol. 147, pp. 195-7, 1981.

[5] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proc Natl Acad Sci U S A*, vol. 85, pp. 2444-8, 1988.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J Mol Biol*, vol. 215, pp. 403-10, 1990.

[7] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Res*, vol. 27, pp. 2369-76, 1999.

[8] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg, "Fast algorithms for large-scale genome alignment and comparison," *Nucleic Acids Res*, vol. 30, pp. 2478-2483, 2002.

[9] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, "Versatile and open software for comparing large genomes," *Genome Biol*, vol. 5, pp. R12, 2004.

[10] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York: Cambridge University Press, 1997.

[11] A. Phillippy, J. Mason, K. Ayanbule, D. Sommer, E. Taviani, A. Huq, R. Colwell, I. Knight, and S. Salzberg, "Comprehensive DNA Signature Discovery and Validation," *PLoS Computational Biology*, In Press.

[12] J. Owens, "Streaming Architectures and Technology Trends," in *GPU Gems 2*, M. Phar, Ed. Upper Saddle River, NJ: Addison-Wesley, 2005, pp. 457-470.

[13] N. K. Govindaraju, L. Scott, G. Jim, and M. Dinesh, "Memory---A memory model for scientific algorithms on graphics processors," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Tampa, Florida: ACM Press, 2006.

[14] Y. Juekuan, W. Yujuan, and C. Yunfei, "GPU accelerated molecular dynamics simulation of thermal conductivities," *J. Comput. Phys.*, vol. 221, pp. 799-804, 2007.

[15] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2005.

[16] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-Sequence Database Scanning on a GPU," presented at 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006) (HICOMB Workshop), Rhode Island, Greece, 2006.

[17] M. Charalambous, P. Trancoso, and A. Stamatakis, "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor," presented at Proceedings of the 10th Panhellenic Conference on Informatics (PCI 2005), Volos, Greece, 2005.

[18] I. Buck, "Taking the Plunge into GPU Computing," in *GPU Gems 2*, M. Phar, Ed.: Addison-Wesley, 2005, pp. 509-519.

[19] "nVidia Compute Unified Device Architecture (CUDA) Programming Guide, version 0.8," 2007.

[20] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings," *Int. J. Parallel Program.*, vol. 29, pp. 217-247, 2001.

[21] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Efficient Tree Layout in a Multilevel Memory Hierarchy," presented at 10th European Symposium on Algorithms (ESA), 2002.

[22] E. Ukkonen, "On-line construction of suffix-trees," *Algorithmica*, vol. 14, pp. 249-260, 1995.