Accelerating Protein Sequence Search in a Heterogeneous Computing System

Shucai Xiao*, Heshan Lin†, and Wu-chun Feng*†

*Department of Electrical and Computer Engineering
†Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24061
Email: {shucai, hlin2, wfeng}@vt.edu

***Abstract*—The "Basic Local Alignment Search Tool" (BLAST) is arguably the most widely used computational tool in bioinformatics. However, the computational power required for routine BLAST analysis has been outstripping Moore's Law due to the exponential growth in the size of the genomic sequence databases that BLAST searches on.**

To address the above issue, we propose the design and optimization of the BLAST algorithm for searching protein sequences (i.e., BLASTP) in a heterogeneous computing system. The end result is a BLASTP implementation that delivers a seven-fold speedup over the sequential BLASTP for the most computationally intensive phase (i.e., hit detection and ungapped extension) on a NVIDIA Fermi C2050 GPU. In addition, when pipelining the processing on a dual-core CPU and the NVIDIA Fermi GPU, our implementation can achieve a six-fold speedup for the overall program execution.

***Index Terms*—BLAST, Graphics Processing Unit (GPU), Sequence Alignment, Parallel Bioinformatics**

I. Introduction

The "Basic Local Alignment Search Tool" (BLAST) [2] is arguably the most widely used tool in bioinformatics. It rapidly identifies the similarities between a set of biological sequences (i.e., nucleotide and protein sequences) and sequence databases. Identified similarities can then be used to infer functional and structural relationships between the corresponding biological entities.

With the exponential growth of biological sequence databases, the computational power required for routine BLAST analysis has been outstripping Moore's Law. For example, in the early 2000s, researchers found that searching a popular public sequence database slowed by 64% each year [5]. Today, next-generation sequencing technologies and emergent research areas in the life sciences, such as metagenomics, promise to accelerate the generation of sequence data to unprecedented rates,[1] thus further exacerbating the gap between the acquisition of sequence data and the ability to compute (i.e., search) on that data. To address this issue, many studies have been conducted in accelerating BLAST on multicore CPUs and accelerators such as the field-programmable gate array (FPGA) or Cell Broadband Engine.

[1]The NIH GenBank contains approximately 100-billion base pairs accumulated over a 27-year lifetime. Today, next-generation sequencers can generate that many base pairs in a few days.

The graphic processing unit (GPU) has evolved into an accelerator with the capability of general-purpose computing. Compared to a traditional CPU, the performance of a GPU has increased at a much faster rate. For instance, the difference in the peak floating-point operations per second (FLOPS) between an NVIDIA GPU and an Intel CPU was about 500 GFLOPS in November 2006; this difference now exceeds 1,200 GFLOPS, as of March 2010. As a consequence, GPU-based systems enjoy increasing popularity in the high-performance computing (HPC) community. In fact, three out of today's five fastest supercomputers are GPU-accelerated, according to the most recent TOP500 List [1]. Although many inherently data-parallel or task-parallel scientific applications report significant speed-ups, it is unclear how well BLAST-like applications, which represent a major type of bioinformatics HPC workloads, will perform on GPUs.

In this study, we present our experience in parallelizing the BLAST search of protein sequences (i.e., BLASTP) on GPUs. Our goal was to identify the opportunities and understand the limitations of accelerating BLAST on the GPU architecture. We found that the BLAST algorithm is highly irregular and does *not* naturally map well onto the GPU. Specifically, BLAST uses a heuristic algorithm, whose execution path and memory-access pattern depend heavily on the similarity between every pair of compared sequences.

To address the above, we first identified performance limitations in accelerating BLAST on GPUs — memory throughput, divergent branching, and load imbalance. We then sought to optimize our BLASTP code on the GPU in order to address these limitations. Finally, we evaluated and characterized the performance implications of our optimizations as well as the aforementioned architectural limitations on two generations of GPUs, i.e., GT 200 and GT 400 (Fermi).

The experimental results show that our GPU implementation of BLASTP, when running on a NVIDIA Tesla Fermi C2050 GPU, can achieve a *seven-fold speedup* over a highly optimized sequential BLASTP running on a CPU for the most computationally intensive phase (i.e., hit detection and ungapped extension). Moreover, when pipelining the processing on a dual-core CPU and a Fermi GPU, our implementation delivers a *six-fold speedup* for the overall program execution.

The rest of this paper is organized as follows. Section II provides a brief background about the CUDA programming model and the BLASTP algorithm. Section III surveys the related work. Sections IV and V discuss our parallelization of the BLASTP and our methodologies in optimizing the program performance, respectively. We then present our performance evaluation and characterization in Section VI and conclude in Section VII.

II. Background

In this section, we give a brief description of the GPU architecture, its associated CUDA programming model, and the BLAST algorithm.

A. GPU Architecture and CUDA Programming Model

Originally, GPUs were designed solely for graphics applications, which are compute-intensive and data-parallel in nature. With the elimination of key architectural limitations, GPUs have evolved from their traditional roots as a graphics pipeline into programmable devices that can support general-purpose scientific computation, i.e., general purpose computation on GPUs (GPGPUs). With the introduction of easy-to-use programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [22] and OpenCL [10], more and more applications continue to be ported to the GPU [8], [18], [20], [25], [27].

A NVIDIA GPU consists of a set of streaming multiprocessors (SMs), where each SM contains a few scalar processors (SPs) On each SM, there are four types of on-chip memory, i.e., register, shared memory, constant cache, and texture cache. This on-chip memory can only be accessed by threads executing on the same SM. On a GPU card, there are also two types of off-chip memory, i.e., global memory and local memory. Global memory can be accessed by all threads on the GPU; while local memory is used in the same way as registers except that it is off-chip.

Within the GPU memory hierarchy, on-chip memory has low access latency but a relatively small size. On the contrary, off-chip memory has a much larger size but also high access latency. One way to improve the efficiency of accessing off-chip global memory is to use *coalesced read/write* operations. On the latest NVIDIA Fermi GPU architecture, L1 and L2 caches are provided to improve the efficiency of global memory access, especially for irregular access patterns.

CUDA is an extension of the C programming language provided by NVIDIA. It allows compute-intensive and data-parallel parts of a program to be executed on a GPU to take the advantage of its computational capability. Specifically, parallel portions of the program are implemented as *kernels* and compiled into device instruction sets. Kernels are called on the host and executed on the device. Each kernel consists of a set of blocks, and each block contains a set of threads.

In addition to the above, CUDA provides functions for read-modify-write atomic operations. We also ensure that all the device memory that is needed on the GPU is allocated In CUDA, there are functions provided for read-modify-write atomic operations. Also, since dynamic memory allocation is *not* supported[2], all device memory should be allocated beforehand. Finally, memory address space on the device is different from that on the host. Consequently, pointers within host-side data structures such as linked list will become invalid after transferred to the device memory.

B. Basic Local Alignment Search Tool

BLAST is actually a family of algorithms, with variants used for searching alignments of different types (i.e., protein and nucleotide) of sequences. Among them, BLASTP is used to compare protein sequences against a database of protein sequences. There are four stages in the BLASTP algorithm:

1) *Hit detection.* Hit detection identifies high-scoring matches (i.e., *hits*) of a fixed length between a query sequence and a subject sequence (i.e., a database sequence).
2) *Ungapped extension*. Ungapped extension determines whether two or more hits obtained from the first stage can form the basis of a local alignment that does *not* include insertions or deletions of residues. The alignments with scores higher than a certain threshold will be passed to the next stage.
3) *Gapped alignment*. This stage performs further extension on the previously obtained alignments with gaps allowed. The result alignments will be filtered with another threshold.
4) *Gapped alignment with traceback.* In this stage, the final alignments to be displayed to users are re-scored, and the alignments are generated using a traceback algorithm.

Figure 1 gives an example of the first three stages of alignment computation. The fourth stage repeats the third one with traceback information recorded. BLAST reports alignment scores calculated based on a *scoring matrix* and *gap penalty factors*. In addition, statistic information such as *"expect" value* that measures the significance of each alignment is also reported.

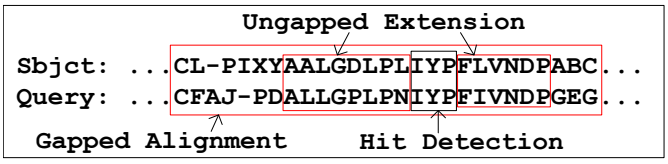


Fig. 1. First Three Stages of BLAST Execution

Our study in this paper is based on FSA-BLAST 1.05 [7], a highly optimized sequential BLAST implementation.

III. Related Work

Since the BLAST tool is both compute- and data-intensive, many approaches have been investigated to parallelize BLAST in the past. On multi-core platforms, the BLAST implementation from National Center for Biotechnology Information (NCBI) has been parallelized with pthreads. On cluster

[2]We noticed that this feature is added in CUDA 3.2. But when this paper was submitted, CUDA 3.2 is unavailable.

platforms, there are parallel implementations such as TurboBLAST [4], ScalaBLAST [24], and mpiBLAST [6], [13], [14]. Among them, mpiBLAST is a widely used parallelization of NCBI BLAST. Combining efficient task scheduling and scalable I/O design, mpiBLAST can effectively leverage tens of thousands of processors to speedup the BLAST search [12].

Parallel BLAST has also been implemented on accelerators such as FPGAs [9], [11], [17], [21], [26], [30]. In a recent study, Mahram et al. [17] introduced a co-processing approach that leverages both the CPU and FPGA to accelerate BLAST. Specifically, their implementation parallelizes the first two stages of BLAST on the FPGA to pre-filter dissimilar subject sequences. Then, the original NCBI BLAST is called on the CPU to search the filtered database. Their implementation can generate the same results as NCBI BLAST and achieve as much as 25-fold performance improvement.

Our work is mostly related to BLAST parallelization on GPUs. Liu et al. [15], [16] developed CUDA-BLASTP and reported a 10-fold speedup over NCBI BLASTP on a desktop machine with two Tesla C1060 GPUs. CUDA-BLASTP uses a pre-filtering design similar to the FPGA study by Mahram et al. [17], and it does not parallelize all the compute stages of the BLASTP algorithm on the GPU. The filtering approach may suffer high overhead when searching BLAST jobs with a large number of subject sequences similar to the query. As we will see in Section VI, our BLASTP implementation on CUDA is two times faster than CUDA-BLASTP. Vouzis et al. [29] introduced another implementation of BLASTP on the GPU. In their implementation, databases are partitioned and processed on both the GPU and CPU, so that the system resources can be better utilized. Their approach also parallelizes only the first two stages on GPUs. With one CPU-helper thread, Vouzis's GPU BLAST implementation achieves between a three- and four-fold speedup for various query sequences.

IV. Mapping BLASTP on CUDA

In this section, we describe how we map the BLASTP algorithm onto the GPU.

A. Profiling of Serial BLASTP

We first profile the execution of BLASTP by searching two sequences (`query1` and `query2`) against the NCBI NR database, which contains 9,874,397 sequences with a total size over 5 GB. The sizes of `query1` and `query2` are 1K and 2K, respectively. Table I shows the time consumed by the four stages for searching the two query sequences. Note that the execution time of the first two stages cannot be separated because these two stages are executed together (details will be described in Section IV-B). Clearly, the first three stages, i.e., hit detection, ungapped extension, and gapped alignment, consume more than 99% of the total execution time, regardless the query sequence length. Thus, our implementations focus on parallelizing the first three stages.

B. Hit Detection and Ungapped Extension Parallelization

As mentioned earlier, the first two stages of BLASTP, i.e., *hit detection* and *ungapped extension* are actually combined together. The algorithm first picks up a word from the beginning of the subject sequence and scans it against the query sequence to find hits. Once a hit is found, the extension is performed immediately on the hit, in both directions. After the extension is done, the algorithm moves on to scan for the next hit in the query sequence that matches the current word of the subject sequence so far and so on. After the current word of the subject sequence is compared against the entire query sequence, the algorithm moves on to the next word in the subject sequence. Since the scanning of a subject word depends on the hit detection and extension results of previous subject words (for more details see [3]), only limited parallelism can be exploited in aligning a pair of sequences in the current BLASTP implementation. Consequently, we parallelize the BLASTP algorithm by having each thread align a pair of sequences (i.e., a query sequence and a subject sequence). In this way, multiple pairs of sequences are aligned concurrently, as shown in Figure 2.

Before the kernel is launched to the GPU, query sequences, subject sequences and a few other data structures are transferred from the host memory to the device memory. Each time, there is one query sequence on the device, which is shared by all threads in the kernel. Also, the database is divided into different chunks, which are searched one after another on the GPU card (one kernel launch per chunk). The chunk size is limited by the global memory size as well as the on-chip memory (as described in Section V-A2) size on a GPU card. By searching one chunk at a time, our GPU implementation can process a database of arbitrary size.

Within a kernel launch, different threads align different subject sequences against the query sequence. When a thread finds successful ungapped extensions, it stores them in the global memory. Since all threads can find ungapped extensions in parallel, care must be taken to avoid write conflicts between different threads. One design option is to have all threads share a global memory buffer, and each thread calls an atomic operation to find a write location for each ungapped extension. Such a design can incur high synchronization overhead because atomic operations are expensive on GPUs. Another design option is to maintain a fixed-size buffer for each thread to store ungapped extensions.[3] This design can avoid the synchronization overhead of atomic operations. However, such a design is space-inefficient because the number of ungapped extensions generated by each thread can differ significantly, and that number cannot be known beforehand.

We propose a *two-level hierarchical* buffer for storing the ungapped extension, as shown in Figure 3. In the level-1 buffer, each thread is allocated a fixed-size segment which can store N ungapped extensions. The level-2 buffer, which can store M ($M >> N$) ungapped extensions, is shared by all threads and guarded with atomic operations. The writing procedure of ungapped extensions is given by Algorithm 1. As can be seen, a thread first writes an ungapped extension to its allocation in the level-1 buffer. When a thread uses up

[3]Recall dynamic memory allocation was not supported on NVIDIA GPUs at the submission time of this paper.

TABLE I
PROFILING OF SERIAL BLASTP (UNIT: SECOND)

Query sequence	Hit detection + ungapped extension	Gapped alignment	Gapped alignment w/ traceback	Total time
Query 1	144.28 (76.09%)	44.87 (23.67%)	0.46 (0.24%)	189.61 (100.00%)
Query 2	260.05 (76.56%)	78.92 (23.23%)	0.70 (0.21%)	339.67 (100.00%)

Note: Numbers in the bracket are percentages of execution time of a stage in the total execution time.

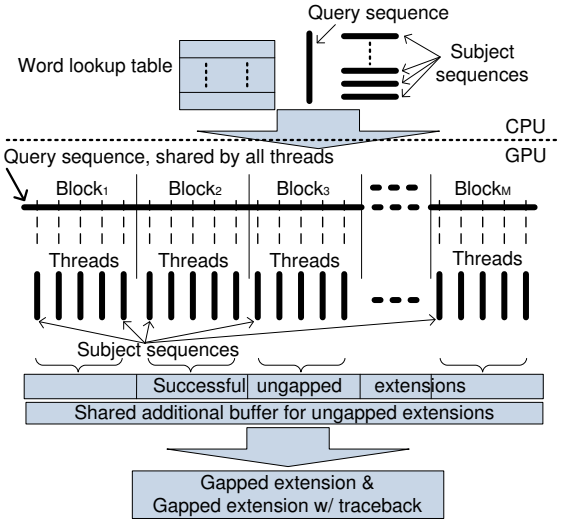


Fig. 2. Hit Detection and Ungapped Extension Parallelization

its allocation in the level-1 buffer, it writes the rest of the ungapped extensions to the level-2 buffer. Such a hierarchical buffering design can efficiently utilize the global memory space as well as avoid unnecessary synchronization overhead.

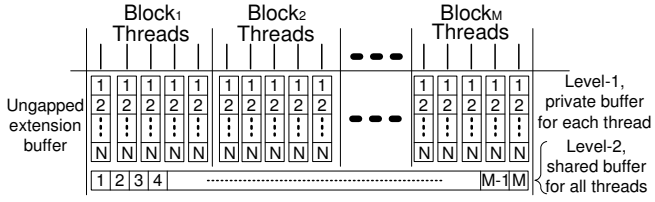


Fig. 3. Ungapped Extension Storage in Global Memory

In the first two stages, the BLASTP algorithm also needs to maintain several global counters, such as the number of hits detected. One way to implement a global counter is to have each thread update a shared variable with atomic add operations. Again, to avoid the overhead of atomic operations, we implement per-thread counters on GPU, and all the per-thread counters will be added up together on the host side to produce the correct value for a global counter.

C. Gapped Alignment Parallelization

Gapped alignment uses seeds created by the ungapped extension stage as its inputs and creates alignments with even higher alignment scores. Typically only a small percentage of database sequences will need to be processed with gapped alignment. As such, we launch a separate kernel for this stage to re-map tasks to individual threads. To minimize data transferring overhead, the gapped alignment kernel reuses the subject sequence data stored on the GPU during the first two stages.

During the gapped alignment, the best alignment score corresponding to each subject sequence is recorded, which will be copied to the host memory to filter out dissimilar subject sequences. In addition, the status of each extension will be recorded and copied back to the host memory for further processing in the final stage – gapped alignment with traceback.

Algorithm 1 Ungapped Extension Storage in Global Memory

```
1: shrdIndex ← 0
2: privtIndex ← 0
3: B ← privtBufSize
4: ..., an ungapped extension is found ...
5: if privtIndex < B then
6:    bufPtr ← privtBuf + privtIndex
7:    privtIndex ← privtIndex + 1
8: else
9:    bufPtr ← shrdBuf + atomicAdd(shrdIndex, 1)
10: end if
11: bufPtr ← unExtPtr
12: ...
```

V. PERFORMANCE OPTIMIZATION

Because of its heuristic nature, the BLASTP algorithm is highly irregular with respect to the memory access and execution path. As such, the basic algorithmic mapping described in Section IV does not map well onto the GPU architecture. In this section, we present several optimization techniques to address some of the performance hurdles of accelerating BLASTP on GPUs.

A. Memory Access

The BLAST search needs to access a number of different data structures. To improve memory access efficacy, we explore different data placement strategies in the GPU memory hierarchy.

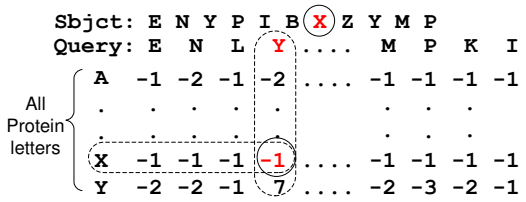
1) Constant Memory to Store the Query Sequence and Scoring Matrix: When calculating alignment scores, the BLASTP algorithm needs to frequently compute a matching score between two individual letters from the query and the subject sequences, which can be done by looking up

the corresponding element in a scoring matrix. FSA-BLAST optimizes the lookup performance by pre-computing a *query profile* for each query sequence. Specifically, a query profile is a two-dimensional matrix, where each column corresponds to one letter in the query sequence and consists of matching scores between the query letter to all other letters as shown in Figure 4(a). With the query profile, the matching score between two letters can be obtained by first finding the column corresponding to the letter in the query sequence and then reading the score from the column according to the letter in the subject sequence. Using query profiles is more efficient because it saves one memory access used to read the current letter of a query sequence.

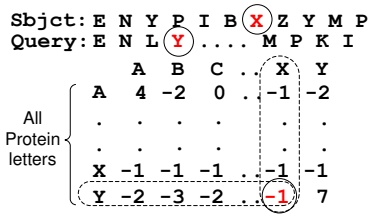
One common optimization technique in GPU programming is to leverage the cached constant memory to speedup the access of frequently used data. Compared to global memory, constant cache has much lower access latency for *cache hits*, but its size is small (64KB). In the FSA-BLAST implementation, each column (corresponding to a letter in the query sequence) in the query profile consumes 64 bytes (32 elements with 2 bytes each). As such, the constant memory is not sufficient to store the query profile for a query sequence larger than 1K letters.

To take advantage of the constant memory, our implementation instead puts the scoring matrix there because the matrix has a fixed size. In fact, the scoring matrix used in BLASTP consists of $32 \times 32 = 1024$ elements and has a total size of only 2KB (2 bytes per element). However, the query sequence needs to be available when using the scoring matrix. In our implementation, a 60K-byte buffer is allocated in the constant memory for its storage. Since one byte is needed for each letter, the maximum query sequence that can be supported is 60K letters. By counting the sequence size in the most recent NCBI NR database, we found that more than 99.95% of the sequences are smaller than 4K letters and the largest sequence contains 36,805 letters, suggesting that the 60K buffer is sufficient for storing individual protein sequences in the most recent NCBI NR database.

Figure 4 depicts the differences between using the query profile and the scoring matrix. Let t_g and t_c be the access latency of global memory and constant memory, respectively, and t_a be the cost of an arithmetic operation. As shown in Figure 4(a), to obtain the matching score between 'X' in the subject sequence and 'Y' in the query sequence, the program should first read 'X' from the subject sequence then read the score '-1' from the query profile, which takes a total time of $2t_g$. In contrast, with the scoring matrix, the program needs to read 'Y' and the score '-1' from the constant memory, and 'X' from the global memory as shown in in Figure 4(b). In addition, two arithmetic operations are needed to compute the location of '-1' in the scoring matrix. Thus, the total time is $t_g+2t_c+2t_a$ in the scoring matrix approach. Since the latency of constant cache is two orders of magnitude faster than global memory and the latency to executemagnitude faster than global memory, using the scoring matrix can be much more efficient than using the query profile if the scoring matrix and the query sequence in the constant memory are well cached. As we will show in Section VI, the experimental results suggest that our approach is effective in practice.



(a) Reading Alignment Score via Query Profile



(b) Reading Alignment Score via Scoring Matrix

Fig. 4. Constant Memory Usage for Individual Alignment Score

2) Texture Memory to Store Subject Sequences and the Word Lookup Table: Texture memory is another type of cached memory on the GPU but with a much larger size than constant memory. For example, with the one-dimensional texture memory, the number of elements that the texture memory can bind to is 2^{27} or 128M elements. In order to take advantage of the texture cache, which has low access latency for cache hits, we partition the database into different chunks of 128MB each. By loading the database chunks on the GPU one after another, our design can search database of an arbitrary size, which is important for solving real-world BLAST search problems in practice.

Storing database sequences in texture memory may also help exploit data locality in alignment computation. For instance, as shown in Figure 5, when a hit is found, ungapped extension will be performed in both directions. With subject sequences stored in texture memory, some portions of the subject sequence may have been cached, thus improving the memory access efficiency.

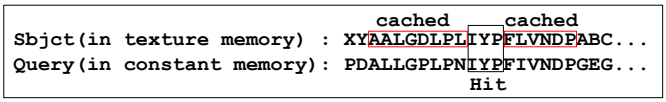


Fig. 5. Texture Memory Usage for Subject Sequences

Besides subject sequences, we also store the *word lookup table* in the texture memory. A word lookup table stores precomputed words that can result in hits to each word in the query profile. Again, the size of the word lookup table varies depending on the query size, and it cannot fit into the constant memory for reasonably long query sequences.

B. Load Balancing across Different Threads

When scheduling alignment tasks to different threads in a kernel, a straightforward implementation can be statically assigning a set of sequences to each thread according to the thread ID number. This approach is easy to implement. However, the overall kernel execution time may suffer when there is load imbalance between different threads.

Algorithm 2 Dynamic Subject Sequence Assignment Algorithm

```
1: n ← totalThreadNum
2: mutexIndex ← n
3: seqIndex ← threadID
4: while seqIndex < numSequences do
5:    AlignSeq(SubSeq[seqIndex], querySeq)
6:    ...
7:    seqIndex ← atomicAdd(mutexIndex, 1)
8: end while
```

To alleviate the load-imbalance issue, our implementation adopts a greedy algorithm (as shown in Algorithm 2) that dynamically assign sequences to different threads. Specifically, the first sequence is assigned to each thread according to the thread ID. Whenever a thread finishes its current assignment, it retrieves the next subject sequence using an atomic operation. In addition, the database sequences will be presorted in the descending order of the sequence lengths. Assuming the BLAST search time is proportional to the length of a subject sequence, the database sorting can alleviate the impact of load imbalance occurred toward the end of the kernel execution. Note that this approach is only applicable to threads in different warps, because threads within a warp always execute the same instructions.

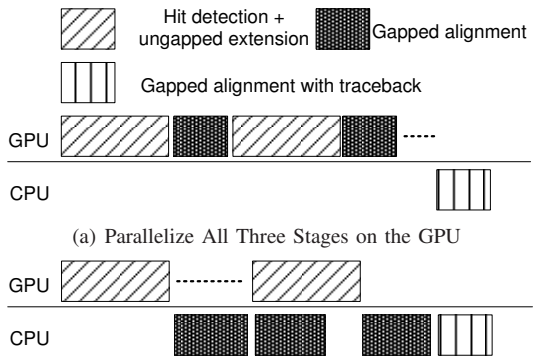
C. Overlap CPU and GPU Computation

In BLAST, the first two stages are executed together, while the execution of the third stage is independent. If we execute all three stages on the GPU, the CPU will be left idle most of the time. To improve resource utilization in the system, we propose to pipeline the procedure by executing the first two stages on the GPU and the third stage on the CPU.

With the pipelining design, the computation on the CPU and GPU can be overlapped as shown in Figure 6. Specifically, the database is divided into multiple chunks. The GPU starts executing the first two stages (i.e., hit detection and ungapped extension) for one chunk, after which the successful ungapped extensions are copied back to the CPU. When the third stage (i.e., gapped alignment) is executed on the CPU on the current trunk, the GPU can start the first two stages for the next chunk. In addition, we use `pthread` to parallelize the gapped alignment in order to leverage the processing power of multi-core CPUs.

VI. Performance Evaluation and Characterization

In this section, we evaluate the performance of our parallel BLASTP implementations on the GPU. Our experiments focus on five versions of BLASTP with different optimization techniques applied. Version 1 is the basic parallel version as described in Section IV. Each of the other four versions is corresponding to an optimization technique discussed in Section V. The five versions are listed in Table II.



(a) Parallelize All Three Stages on the GPU

(b) Overlap CPU and GPU Computation

Fig. 6. Two Methods for Gapped Alignment

TABLE II
Versions of GPU BLASTP

Versions	Description
Version 1	It is a straightforward mapping as described in Section IV.
Version 2	Constant memory is used as described in Section V-A1.
Version 3	Based on Version 2; atomicAdd is called for load balancing as described in Section V-B.
Version 4	Based on Version 2; texture memory is used as described in Section V-A2.
Version 5	Based on Version 4; atomicAdd is called for load balancing as described in Section V-B.

Our experiments are executed on both the NVIDIA Tesla C1060 and the Tesla C2050 GPU cards. The C1060 GPU consists of 30 SMs, each containing 8 scalar processors. On each SM, there are 16K registers and 16KB shared memory. There is 4GB of global memory on the C1060 with an aggregate bandwidth of 102.4GB/s. The C2050 is a newer-generation GPU from NVIDIA. Compared to C1060, C2050 has more scalar processors (i.e., 32) per SM and larger register files (i.e., 32KB). The C2050 has a L1 cache for each SM and a L2 cache shared by all SMs. Both L1 cache and shared memory use the same on-chip memory, which can be configured as 16 KB L1 cache and 48KB shared memory or as 48KB shared memory and 16 KB L1 cache. The L2 cache has a larger size of 768KB. There is 3GB global memory with an aggregate memory bandwidth of 153.6GB/s on the C2050. We will refer the C1060 and C2050 as *Tesla* and *Fermi*, respectively, hereafter.

On the host side, the system has an Intel Core 2 Duo CPU with 2.2GHz clock speed and 4GB DDR2 SDRAM memory. The operating system is the Ubuntu 8.04 GNU/Linux distribution. Our code is developed with the CUDA 3.1 toolkit.

We use a subset of the NCBI NR database[4] with one sequence selected out of every 5 sequences from the NR database. Also, four sequences with 1K, 2K, 3K, and 4K letters, respectively, are used as the query sequences.

For the BLASTP program, all default values are used for the program execution as shown in Table III. In addition, the score matrix BLOSUM62 is used. The serial CPU version is compiled with gcc with the -O3 optimization option. We report the average number of three runs for each experiment.

TABLE III
DEFAULT PARAMETER VALUES IN BLASTP

Parameter description	Value
Word size	3
Dropoff value for ungapped extension	7
Dropoff value for gapped extension	15
Dropoff value for triggering gaps	22
Dropoff value for final gapped alignment	25
Open gap penalty	-7
Extension gap penalty	-1

A. Evaluation of Individual Optimization Techniques

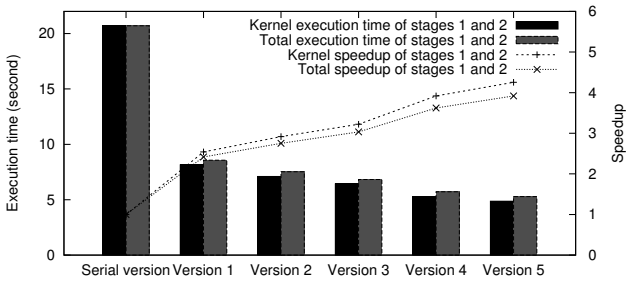
In this section, we evaluate the performance impacts of individual optimization techniques described in Section V with respect to the execution time spent on various compute stages.

1) Hit Detection + Ungapped Extension: Figure 7 shows the execution time of the first two stages for the five versions as described in Table II as well as the baseline serial CPU version. The 1K query sequence is used as the input. We measure the kernel execution time and total execution time of the first two stages, where the total execution time includes data transfer time between host memory and device memory, pre/post-processing time, and kernel execution time[5]. We also calculate the speedups of various GPU versions against the CPU version.

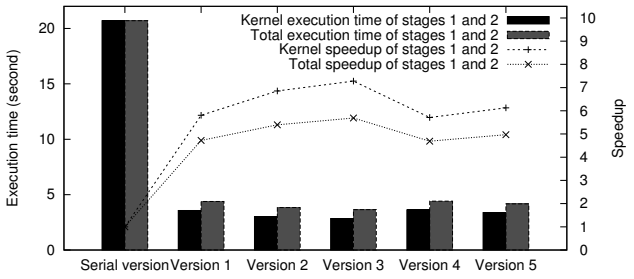
Figure 7(a) shows the results on the Tesla card. Clearly, performance improvements are achieved when each optimization technique is applied. Specifically, the kernel execution time is 8.148s in Version 1, while that of Version 2 is 7.098ms, which means a performance improvement of 12.89% is achieved when the query sequence and scoring matrix are stored in the constant memory. With the load-balancing optimization added, i.e., Version 3, the kernel execution time is further reduced to 6.422s, a 9.51% improvement compared to Version 2. Version 4 extends Version 2 by placing the subject sequences and the word lookup table in the texture memory, resulting a 25.55% performance improvement. Finally, the best performance is achieved in Version 5, where load balancing is added as compared to Version 4. For the total execution time, the absolute differences between different versions are the same as those for the kernel execution time. With each optimization technique applied, the relative performance improvements are

[4]According to our experiments with various databases with different sizes, the speedup achieved is quite stable regardless of the database size. Therefore, we use a subset of the NR database in our experiment.

[5]Since there is no data transfer, pre-processing, and post-processing in the serial CPU version, the kernel execution time and the total execution time is the same for the serial CPU version.



(a) Tesla Results



(b) Fermi Results

Fig. 7. Performance Improvements Brought by each Optimization Technique and the Corresponding Speedup for the First Two Stages

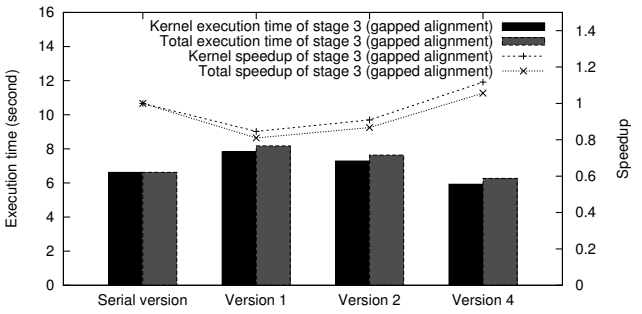
12.18% (Versions 1 to 2), 24.04% (Versions 2 to 3), 9.29% (Versions 2 to 4), and 7.56% (Versions 4 to 5), respectively.

On Fermi, similar trends are observed for Versions 1, 2 and 3 as compared to the Tesla results. However, storing the subject sequences and the word lookup table in the texture memory has adverse performance impacts on Fermi. Specifically, Version 4 is 20.04% slower than Version 2. This can be caused by the L1/L2 cache introduced in the Fermi architecture, which will be explained in more details in Section VI-D1. Nonetheless, with load balancing added, Version 5 outperforms Version 4 by 6.80%, similar to what we observed on Tesla.

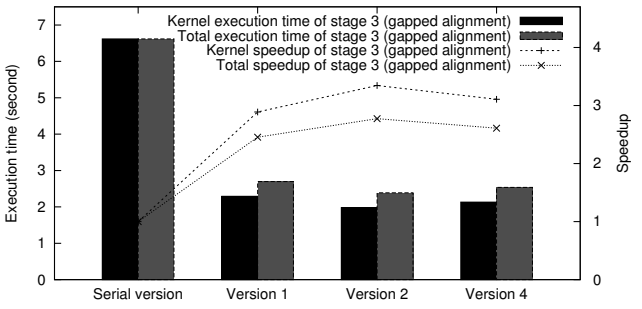
Compared to the CPU serial version, the best GPU versions achieve speedups of 4.25 and 7.28 on Tesla and Fermi, respectively, in kernel execution time. For the total execution time, the speedups are 3.92 and 5.69 folds on Tesla and Fermi, respectively. As expected, the program performance on Fermi is better than that on Tesla. One reason is that the first two stages of BLAST are memory-bound, and Fermi has a larger global memory bandwidth. Moreover, there are L1 and L2 caches on the Fermi card, which can improve the efficacy of global memory access even more.

2) Gapped Alignment: This section shows the performance improvements of the gapped alignment stage. Since each thread is responsible for only one subject sequence, the optimization approach by using atomic function to achieve load balance cannot be used here. Thus there are three different GPU implementations for the parallelization of the gapped alignment, which are Versions 1, 2, and 4 as described in Table II. Again, the 1K query is used as in the previous section.

As shown in Figure 8(a), on Tesla, performance improve-



(a) Tesla Results



(b) Fermi Results

Fig. 8. Performance Improvements Brought by each Optimization Technique and the Corresponding Speedup for the Gapped Alignment Stage

ments are observed with each optimization technique applied, similar to what is observed for the first two stages. With the constant memory used for the query sequence and the scoring matrix, performance of the gapped alignment is improved by 6.87% and 6.60% for the kernel and total execution time, respectively. Storing the subject sequences in the texture memory help reduce the kernel and total execution time by 18.68% and 17.87%, respectively. Unfortunately, there is no performance improvement for the gapped alignment on the Tesla compared to CPU serial implementation. One reason can be that the irregular memory access is poorly supported on Tesla. Another reason can be the large number of divergent branches in the gapped alignment code. We will leave further investigations of this issue for the future work.

Figure 8(b) shows the performance results on Fermi. With constant memory used to store the query sequence and the scoring matrix, the kernel execution time decreases from 2.291s (Version 1) to 1.979s (Version 2), corresponding to a 13.61% improvement. However, if the subject sequences are stored in the texture memory, the kernel execution time is increased by 7.63% to 2.130s (Version 4). As we discussed above for the first two stages, the reason is that the global memory access is more efficient on Fermi because of the L1/L2 cache provided. The best GPU version achieves 3.34-fold and 2.77-fold speedups for kernel and total execution time, respectively, as compared to the serial version on CPU.

It is worth noting that on both Tesla and Fermi, the kernel execution time occupies a majority (more than 80%) of the total execution time in each stage (Hit detection and ungapped extension are measured together.) as shown in Table IV.

TABLE IV
PERCENTAGE OF THE KERNEL EXECUTION TIME

Stage(s)	Tesla	Fermi
Hit detection + ungapped extension	92.08%	80.36%
Gapped alignment	94.53%	84.15%

3) Overall Execution Time: In this experiment, we compare the overall execution time for five different implementations.

- CPU serial implementation.
- *Version G1:* All three stages are executed on GPU.
- *Version G2:* The first two stages are executed on GPU, and the third stage is serially executed on CPU. There is no overlap between the CPU and GPU processing.
- *Version G3:* The first two stages are executed on GPU, and the third stage is executed on the CPU in parallel with two threads. No overlap exists between the CPU and GPU processing.
- *Version G4:* The first two stages are executed on GPU, and the third stage is executed on the CPU in parallel with two threads. The CPU and GPU processing is overlapped.

Figure 9 shows the overall execution time of the above five implementations. There are several observations we can make from Figure 9: First, if all three stages are executed on the GPU, the overall performance on Fermi is much better than that on Tesla (by 1.93 times). Second, with GPUs used for only the first two stages, on Fermi, if no overlap is used (Versions G2 and G3), the overall performance (9.92s and 7.00s for Versions G2 and G3, respectively) will be worse than that of Version G1 (6.10s). On the other hand, if we overlap the computation on the CPU and the GPU, the overall performance (4.98s) will become better than Version G1 (6.10s) by 18.29%. Third, on Tesla, since there is almost no performance improvement by parallelizing the gapped alignment on the GPU, any parallelization or optimization performed for the gapped alignment will bring performance improvements. For instance, if the `pthread` is used for the parallelization, we can reduce the execution time by 27.79% (from Version G2 to Version G3). Furthermore, if the CPU and GPU execution is overlapped, this performance improvement can almost be doubled, i.e., the execution time decreases from the 11.89s of Version G1 to 5.95s of Version G4.

B. Scalability of the Query Size

In this section, we evaluate the scalability of overall program performance with respect to the size of a query sequence. Specifically, we use the four aforementioned query sequences with sizes from 1K to 4K as the input. We show only the results on Fermi because the program performance on Fermi always outperforms that on Tesla.

Figure 10 plots the scalability results in both overall execution time and the corresponding speedup. We compare three different implementations: CPU-only, GPU-only, and the pipelining implementation that overlaps processing on the CPU and GPU as described in Section V-C (the gapped

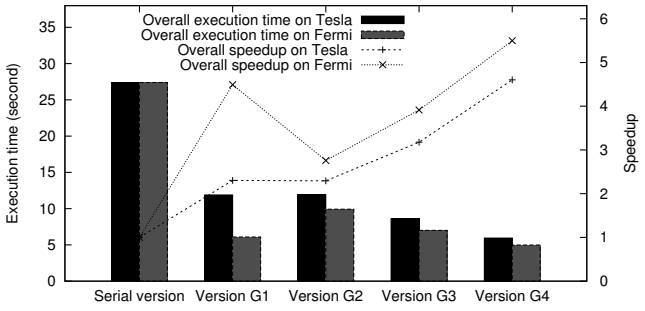


Fig. 9. Overall Execution Time

alignment is parallelized on the CPU using `pthread`). As can be seen from the speedup curves, both the GPU-only and pipelining implementations scale well as the query size increases. The GPU-only implementation can achieve around 4.5-fold speedup for all input queries. Leveraging the processing power of both the CPU and the GPU, the pipelining version can deliver more-than-6-fold speedups for 3K and 4K queries.

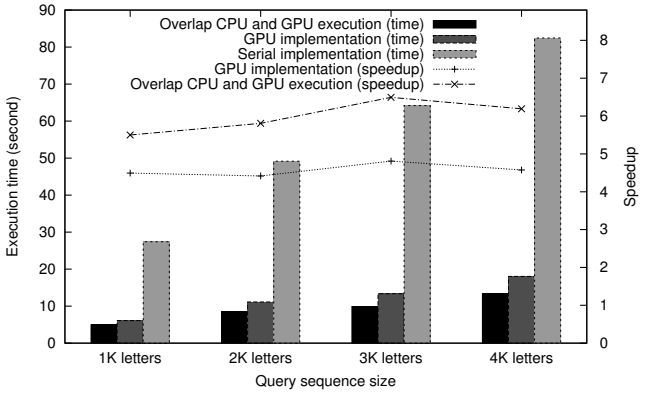


Fig. 10. Scalability of Overall Performance

C. Comparison with the Existing CUDA-BLAST Implementation

As described in Section III, there is an existing GPU implementation of BLASTP by Liu et al. [15][6]. A close examination on the source codes suggests that both Liu's and our kernel implementations are based on the same sequential code. As such, we compare our implementation with the Liu's in this section.

As mentioned before, in the implementation of Liu et al., the GPU is used to only filter subject sequences that yield successful ungapped extensions. After the filtering is done on the GPU, the entire BLASTP program will be executed on the CPU again to search the filtered subject sequences. In other words, only the first two stages are parallelized on the GPU in Liu's implementation.

[6]The source code can be downloaded from http://www.nvidia.com/object/blastp_on_tesla.html.

With the above filtering design, there is redundant computation on the CPU for the first two stages of filtered sequences. This redundant implementation will increase with the amount of qualified subject sequences (i.e., sequences with successful ungapped extensions). Also, the third and fourth stages are not parallelized at all. In the extreme case, where successful ungapped extensions are found for all subject sequences, the Liu's implementation will be much slower than the serial BLAST on CPU. In contrast, our implementation avoid the redundant computation by parallelizing three stages of the algorithm.

We compare our and Liu's implementations by searching two different query sequences; the first (`query 1`) is from the sequence database and the second (`query 2`) is not. Intuitively, the search of `query 1` will generate more qualified subject sequences after the first two stages. The performance results are shown in Figure 11. Note that these results are collected on the Tesla card because Liu's implementation cannot be executed on the Fermi card. As can be seen, our implementation significantly outperforms the Liu's for both query sequences. Also, the performance speedup of our implementation against Liu's implementation is higher for `query 1` (2.01 folds) than for `query 2` (1.83 folds). The reason is that with more qualified subject sequences generated in the first two stages, the redundant computation increases in Liu's implementation.

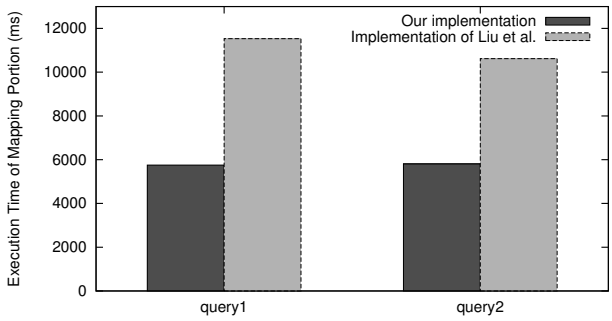


Fig. 11. Performance Comparison between Our Implementation and That of Liu et al.

D. Performance Characterization

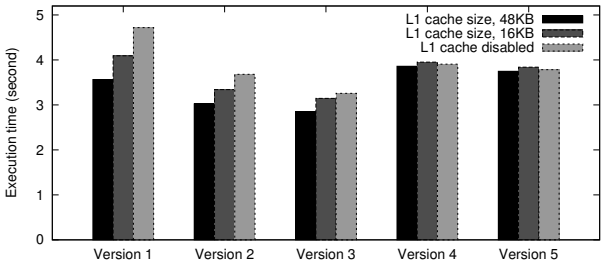
Despite our optimizations, the performance speedup of BLASTP on the GPU is relatively low compared to other applications [23], [28] that map well on the GPU architecture. In this section, we explore the factors that potentially limit the BLASTP performance on GPU through detailed performance characterization.

1) Caching Effects: Starting from Fermi, NVIDIA introduces L1 and L2 memory cache. L1 cache has a configurable size (16KB or 48KB) and is shared by all threads on an SM. L2 cache has a size of 768KB, but it is shared by all SMs. We are interested in the answers of the following questions in characterizing the memory access performance:

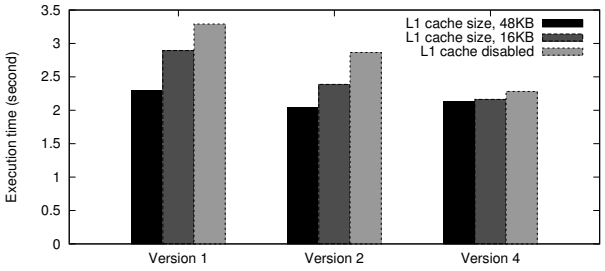
- To what extent can the caching help improve the program performance? To address this question, we will compare

the program performance of three different configurations: 1) disabling L1 cache, 2) using 16KB for L1 cache, and 3) using 48KB for L1 cache [7].

- How does the transparent caching compare to manual memory-access optimizations? Specifically, before Fermi, many memory optimizations are done by leveraging constant and texture cache. We would like to see how these explicit memory placement optimizations compared to relying on the caching mechanism available on the newer-generation of GPU architectures (e.g., Fermi).



(a) Hit Detection + Ungapped Extension



(b) Gapped Alignment

Fig. 12. Impacts of Caching on Kernel Execution Time

Figure 12(a) shows the kernel execution time of the first two stages under various configurations. For versions 1, 2, and 3, the L1 cache size has considerable impacts on the performance. Specifically, when L1 cache size is changed from 48KB to 16KB, kernel execution time of Version 1 increases from 3.57s to 4.09s, corresponding to a relative increase of 14.77%. In addition, when L1 cache is disabled, an additional increase of 15.31% is observed. Similar trends are observed for Versions 2 and 3, but the performance differences caused by the caching effects are in a smaller range compared to Version 1. In contrast, the L1 cache sizes does not have obvious impacts for Versions 4 and 5. One possible reason is that in those two versions, the subject sequences are stored in texture memory, thus reducing the amounts of memory access from the global memory.

Unlike on the Tesla card, where storing subject sequences in the texture memory brings nontrivial performance improvements, on the Fermi card, leaving subject sequences in the global memory actually results in better performance. Specifically, Version 2 outperforms Version 4 by 21.69% when 48KB L1 cache is used. Similar performance improvement is observed for Version 3 over Version 5. One possible reason is that the L2 cache of global memory smoothes out irregular memory access to some extent. The texture cache, which is designed for 2D spatial locality, is less effective than global memory cache in dealing with the memory access patterns in BLAST. This suggests that the caching mechanism introduced in the new Fermi architecture can potentially ease the efforts of optimizing memory access for BLAST-like applications.

Figure 12(b) shows the performance impacts of caching on the stage of gapped alignment. Again, when subject sequences are stored in global memory, the L1 cache size can considerably affect the performance. Specifically, for Version 1, using 48 KB L1 cache results in a 30% improvement compared to the case where L1 cache is disabled. In Version 4, where subject sequences are stored in texture memory, using 48 KB L1 cache still outperforms the non-L1-cache case, but with a much smaller margin (i.e., 6%). Interestingly, storing subject sequences in the texture memory does help when 16KB or less L1 cache is used. This may be attributed to the different memory-access patterns between the first two stages and the third stage. In summary, provisioning larger cache for global memory may be helpful in improving the BLAST performance.

2) Divergent Branching: Because of the heuristic nature of the BLAST algorithm, there are a lot of inherent divergent branches in the code. Since these divergent branches cannot be eliminated in the current GPU architecture, our study on this perspective is to quantify the performance impact of divergent branching on the BLAST performance. Specifically, we synthesize a pseudo input data set, where all the subject sequences in the database are the same[8]. When searching this database, no divergent branches will occur. The performance speedup that can be achieved by this input data set can shed some lights on how much performance impacts are caused by divergent branches. This can also allow us to assess to what extent the program performance is bounded by the memory access throughput.

Tables V compares the performance of searching the synthesized database against the aforementioned NR subset, using the 1K query mentioned earlier as input. Here, the speedup against the CPU serial implementation is shown for each scenario. We use Version 5 on the Tesla card and Version 3 for the Fermi card, because these versions deliver the best performance among others. As can be seen, higher speedups can be obtained (from 5.69-fold to 9.07-fold for Fermi and from 4.25-fold to 7.74-fold for Tesla) by searching the synthesized DB than searching the NR subset. This suggests divergent branching is a major factor that limits the BLAST performance on GPU. Thus, to better support irregular applications like BLAST on GPUs, more effective mechanisms are needed to reduce the impacts of divergent branching and load imbalance. Moreover, even with most of the divergent branches eliminated, the performance speedup is still within 10, suggesting that the program performance is hampered by the efficiency of memory

[7]L2 cache cannot be controlled by programmers on Fermi.

[8]The sequence is randomly selected from the NR database with a size equal to the average sequence size in NR.

access as well.

TABLE V
DIVERGENT BRANCHING IMPACT ON OVERALL SPEEDUP

	Fermi speedup	Tesla speedup
Synthesized DB	9.07	7.74
Subset of NR DB	5.69	4.25

VII. CONCLUSION

Accelerating the BLAST algorithm is of great importance to computational biology. In this work, as a complement of the existing parallel BLAST implementations on multi-core and/or distributed systems, we parallelize BLAST on GPUs to accelerate its execution.

We found that there are many irregularities in both the computation and memory accesses for the execution of BLAST on GPUs, which should be overcome as much as possible to achieve good performance. To address the irregularities and improve performance, we propose techniques including storing query sequences and the scoring matrix in the constant memory, using texture memory to cache subject sequences and the word lookup table, and dynamically assigning sequences to threads to achieve good load balance. Moreover, we overlap the first two stages on the GPU and the third stage on CPU, which is parallelized with `pthread`, and better performance are achieved than that by executing all three stages on GPUs. Compared to the serial CPU implementation, our parallel implementation achieves about 6 times speedup for overall program performance. In addition, we characterize the performance impacts of our parallelization approaches with respect to the cache configuration and divergent branching.

In the future, we will investigate alternative parallelization approaches by relaxing the sequential constraint in the current BLASTP CPU implementation. We would also like to parallelize other BLAST algorithms, e.g., BLASTN, on the GPU. In addition, we plan to leverage the message passing interface (MPI) [19] to implement a multi-level parallelization of BLAST on heterogeneous clusters.

ACKNOWLEDGEMENT

This work was supported in part by NSF grants CNS-0915861 and CNS-0916719 and a NVIDIA Professor Partnership Award.

REFERENCES

[1] Top500 Supercomputer Sites, November 2010. http://www.top500.org/lists/2010/11.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. In *Journal of Molecular Biology*, volume 215, pages 403–410, October 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[4] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. TurboBLAST : A Parallel Implementation of Blast Built on the Turbohub. In *Proc. of the Parallel and Distributed Processing Symposium*, April 2002.

[5] M. Cameron, H. E. Williams, and A. Cannane. Improved Gapped Alignment in BLAST. In *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, volume 1, pages 116–129, July 2004.

[6] A. E. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *Proc. the 4th International Conference on Linux Clusters*, June 2003.

[7] FSA-BLAST. Get FSA-BLAST at SourceForge.net. http://sourceforge.net/projects/fsa-blast/.

[8] W. W. Hu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[9] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. In *ACM Transactions on Reconfigurable Technology and Systems*, volume 1, June 2008.

[10] Khronos OpenCL Working Group. The OpenCL Specification. December 2008. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[11] D. Lavenier. G.: Seed-based Genomic Sequence Comparison Using a FPGA/FLASH Accelerator. In *Proc. of the IEEE Conference on Field Programmable Technology*, 2006.

[12] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. Feng. Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, 2008.

[13] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*. IEEE Computer Society, 2005.

[14] H. Lin, X. Ma, W. Feng, and N. F. Samatova. Coordinating computation and i/o in massively parallel sequence search. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2010.

[15] W. Liu. CUDA-BLASTP on Tesla GPUs. January 2010. http://www.nvidia.com/object/blastp_on_tesla.html.

[16] W. Liu, B. Schmidt, and W. Mueller-Wittig. CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2011.

[17] A. Mahram and M. C. Herbordt. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering. In *Proc. of the 24th ACM International Conference on Supercomputing*, June 2010.

[18] S. A. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, March 2008.

[19] Message Passing Interface Forum. The Message Passing Interface (MPI) Standard. http://www.mcs.anl.gov/research/projects/mpi.

[20] Y. Munekawa, F. Ino, and K. Hagihara. Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU. In *Proc. of the 8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, October 2008.

[21] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: Towards a Portable, Cost-effective Open Source Hardware Implementation. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[22] NVIDIA. NVIDIA CUDA Programming Guide-3.1, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[23] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. *GPU Gems*, 3:677–695, 2007.

[24] C. Oehmen and J. Nieplocha. ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Trans. Parallel Distrib. Syst.*, 17(8), 2006.

[25] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. W. Hwu. GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications. In *Proc. of the Conference on Computing Frontiers*, pages 273–282, May 2008.

[26] E. Sotiriades and A. Dollas. A General Reconfigurable Architecture for the BLAST Algorithm. In *Journal of VLSI Signal Processing*, 2007.

[27] G. M. Striemer and A. Akoglu. Sequence Alignment with GPU: Performance and Design Challenges. In *IPDPS*, May 2009.

[28] V. Volkov and J. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[29] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. In *Bioinformatics*, pages 182–188, 2011.

[30] F. Xia, Y. Dou, and J. Xu. Families of FPGA-Based Accelerators for BLAST Algorithm with Multi-seeds Detection and Parallel Extension. In *Communications in Computer and Information Science*, volume 13, 2008.