cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU

Jing Zhang*, Hao Wang*, Heshan Lin*, and Wu-chun Feng*†
*Dept. of Computer Science and †Dept. of Electrical & Computer Engineering | Virginia Tech
Email: {zjing14, hwang121, hlin2, wfeng}@vt.edu

***Abstract*—BLAST, short for Basic Local Alignment Search Tool, is a fundamental algorithm in the life sciences that compares biological sequences. However, with the advent of next-generation sequencing (NGS) and increase in sequence read-lengths, whether at the outset or downstream from NGS, the exponential growth of sequence databases is arguably outstripping our ability to analyze the data. Though several recent studies have utilized the graphics processing unit (GPU) to speedup the BLAST algorithm for searching protein sequences (i.e., BLASTP), these studies used coarse-grained parallel approaches, where one sequence alignment is mapped to only one thread. Moreover, due to the irregular memory access patterns in BLASTP, there remain significant challenges to map the most time-consuming phases (i.e., hit detection and ungapped extension) to the GPU using a fine-grained multithreaded approach.**

To address the above issues, we propose cuBLASTP, an efficient fine-grained BLASTP implementation for the GPU using CUDA. Our cuBLASTP realization encompasses many research contributions, including (1) memory-access reordering to reorder hits from column-major order to diagonal-major order, (2) position-based indexing to map a hit with a packed data structure to a bin, (3) aggressive hit filtering to eliminate hits beyond the threshold distance along the diagonal, (4) diagonal-based parallelism and hit-based parallelism for ungapped extension to extend sequences with different lengths in databases, and (5) hierarchical buffering to reduce memory-access overhead for the core data structures. The experimental results show that on a NVIDIA Kepler GPU, cuBLASTP delivers up to a 5.0-fold speedup over sequential FSA-BLAST and a 3.7-fold speedup over multithreaded NCBI-BLAST for the overall program execution. In addition, compared with GPU-BLASTP (the fastest GPU implementation of BLASTP to date), cuBLASTP achieves up to a 2.8-fold speedup for the kernel execution on the GPU and a 1.8-fold speedup for the overall program execution.

***Keywords*-BLAST, BLASTP, GPU, bioinformatics, life sciences, next-generation sequencing, hit detection, ungapped extension.**

I. Introduction

The "Basic Local Alignment Search Tool" (BLAST) [3] is a fundamental algorithm in the life sciences that identifies the most similar sequences from the database for a given query sequence. The similarities identified by BLAST can be used to infer functional and structural relationships between the corresponding biological entities. With the advent of next-generation sequencing (NGS) and the increase in sequence read-lengths, whether at the outset or downstream from NGS, the exponential growth of sequence databases is arguably outstripping our ability to analyze the data. Consequently, there have been significant efforts in accelerating sequence-alignment tools on various parallel architectures in recent years.

Graphics processing units (GPUs) offer the promise of accelerating bioinformatics tools due to their superior performance and energy efficiency. Despite the promising speedups that have been reported for other sequence alignment tools such as Smith-Waterman [14], BLAST remains the most popular sequence analysis tool while also being one of the most challenging ones to accelerate on GPUs. Due to its popularity, the BLAST algorithm has been heavily optimized for CPU architectures over the past two decades. These CPU-oriented optimizations can create many obstacles when accelerating BLAST on GPU architectures. First, to improve computational efficiency, BLAST employs input-sensitive heuristics to quickly eliminate unnecessary search space. Although this technique is very effective compared to alignment algorithms that search the entire alignment space, e.g., Smith-Waterman, it makes the program execution path unpredictable, thus easily creating many divergent branches on GPU architectures. Second, to improve memory-access efficiency, the data structures used in BLAST are finely tuned to leverage the large CPU cache. Simply reusing these data structures on GPUs can cause serious inefficiency in memory access because the cache space on GPUs is significantly smaller than that on CPUs.

State-of-the-art BLAST implementations on GPUs [16], [17], [15], [9] all adopt an embarrassingly parallel approach, where one sequence alignment is scheduled to only one thread. In contrast, a fine-grained mapping approach, e.g., using warps of threads to accelerate one sequence alignment, could theoretically better leverage the abundant parallelism offered by GPU architectures. Our experience, however, shows that designing a fine-grained mapping approach for BLAST is very difficult, mainly because of the high irregularity in execution paths and memory-access patterns that are caused by various CPU optimizations of the BLAST algorithm. Further performance improvements for accelerating BLAST on GPUs will require a fundamental rethinking of algorithm design.

Consequently, we propose cuBLASTP, a novel mapping of the BLAST algorithm onto a GPU. First, we decouple

stages in the BLAST algorithm and parallelize stages having different computational patterns with different strategies on the GPU. Second, in order to eliminate the branch divergence and irregular memory access, we propose binning optimizations, including four important techniques: (1) memory-access reordering to reorder hits from column-major order to diagonal-major order, (2) position-based indexing to map a hit with a packed data structure to a bin, (3) aggressive hit filtering to eliminate hits beyond the threshold distance along the diagonal, and (4) diagonal-based parallelism and hit-based parallelism for ungapped extension to extend sequences with different lengths in databases. Furthermore, we design a hierarchical buffering mechanism for the core data structures, e.g., deterministic finite automation (DFA) and position-specific scoring matrix (PSS matrix), with the latest features provided by the NVIDIA Kepler architecture, e.g., read-only cache, to improve the data access bandwidth on GPU.

The experimental results show that when running the most critical phases of BLASTP, i.e., hit detection and ungapped extension, on a NVIDIA Kepler GPU, cuBLASTP achieves up to a 7.8-fold speedup over the highly optimized sequential FSA-BLAST and 2.9-fold speedup over the multithreaded NCBI-BLAST on a quad-core CPU. Moreover, for the overall program execution, cuBLASTP gets up to a 5.0-fold and 3.7-fold speedup, respectively, over sequential FSA-BLAST and multithreaded NCBI-BLAST. This is achieved, in part, by overlapping the data transfer and the kernel execution. In addition, compared with GPU-BLASTP, the fastest GPU implementation of BLAST to date, cuBLASTP achieves up to 2.8-fold speedup for the kernel execution and 1.8-fold speedup for the overall program execution.

II. Background

A. Basic Load Alignment Search Tool

BLAST is a family of algorithms with variants used for different searching alignments, e.g., protein and nucleotide. BLAST algorithms provide the approximation to the dynamical programming method (i.e., Smith-Waterman algorithm). Instead of comparing entire sequences, BLAST algorithms locate high scoring short matches (i.e., hits) between the query sequence and the subject sequences, and extend hits to longer alignments. Only having the slight loss of the accuracy, BLAST algorithms can be significantly faster than Smith-Waterman algorithm. Among them, BLASTP is used to compare protein sequences.

We take FSA-BLAST [1], which has been optimized on CPU for protein sequence searching, as an example to introduce the BLAST algorithm. There are four stages in BLAST algorithm: (1) **hit detection** identifies high scoring short matches (i.e., hits) with a fixed length between a query sequence and the subject sequences; (2) **ungapped extension** determines whether two or more hits from the hit detection can form the basis of a local alignment without insertions and deletions of residues and passes hits with the requested scores to the next stage; (3) **gapped extension** performs the further extension based on alignments from the previous stage and allows gaps; (4) **gapped alignment with traceback** re-scores all alignments from the previous stage using a traceback algorithm and displays the alignments with high scores.

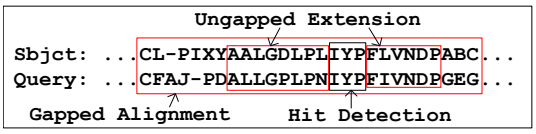


Figure 1: First Three Stages of BLAST Execution [17]

Figure 1 illustrates the first three stages of alignment computation. Since the hit detection and ungapped extension are most compute intensive stages, most of the BLAST optimizations focus on these two stages. In the hit detection and ungapped extension, the core data structures, such as the deterministic finite automaton (DFA) [6], the position-specific scoring matrix (PSS matrix or PSSM) and the scoring matrix, are described as below.

- **Deterministic Finite Automaton (DFA)**, also known as deterministic finite state machine (DFSM), provides a general method for searching one or more fixed- or variable-length strings expressed in arbitrary, user-defined alphabets. In BLAST, the query sequence is decomposed into fixed-length short words and converted into a DFA. As shown in Figure 2(a), each word from a subject sequence goes through the transition of states in DFA. The matched positions will be obtained after a fixed number of transitions.
- **Position-Specific Scoring Matrix (PSS matrix)**, also known as position-specific weight matrix (PSWM), is a type of scoring matrix built from the query sequence. As shown in Figure 2(b), a column in PSS matrix represents a position in a query sequence, and the scores in rows indicate the similarity of all symbols (i.e., amino acid) to the symbol in the column of the query sequence. To obtain the score for 'X' in the subject sequence and 'Y' in the query sequence, the program should get the character 'X' in the subject sequence based on the column number, and go to the row for 'X' with the column number to get the score '-1'. Through checking the PSS matrix, BLAST algorithm can quickly get the similarity between two symbols in corresponding positions of two sequences.
- **Scoring Matrix** is an alternative data structure of the PSS matrix. The scoring matrix has the fixed but smaller size than that of the PSS matrix, since the elements in columns represent words instead of positions in the PSS matrix. The drawback to use the scoring matrix is that more memory access operations are needed. As shown in Figure 2(c), to compare the same pair of letters as above, the program has to load the letter 'X' from the subject

sequence and 'Y' from the query sequence, and then to find out the score '-1' in the column 'X' and row 'Y'.

(a) Read Matched Query Position via DFA [6]

(b) Scoring via PSS Matrix [17]

(c) Scoring via Scoring Matrix [17]

Figure 2: Core Data Structures in BLAST

B. GPU Architecture and CUDA Programming Model

Due to their superior performance, GPUs have been widely used for compute- and data-intensive applications. Since we carry out our evaluations on NVIDIA GPUs, we refer to NVIDIA GPUs architecture and CUDA programming model in the remaining sections.

A NVIDIA GPU consists of a set of streaming multiprocessors (SMs), each of which consists of multiple cores as Single Instruction Multiple Data (SIMD) units. There are two types of memory in GPU, on-chip memory and off-chip memory. On-chip memory, such as register, shared memory, constant cache, etc., has low access latency but a relatively small size. Off-chip memory, including global memory and local memory, has much larger size but higher access latency. To efficiently access data in the global memory, the read/write operations are required to be coalesced. The latest NVIDIA Kepler architecture also offers various caches to improve the efficiency of global memory access, especially for those with irregular access patterns. A 48 Kilobytes cache known as the read-only cache is introduced to improve irregular memory access performance.

Compute Unified Device Architecture (CUDA) [12] is the programming model provided by NVIDIA. The CUDA functions that will be running on GPUs are called GPU kernels. A kernel will be running in parallel by a large number of threads on GPU. The threads are grouped into blocks of threads and grids of blocks. When a kernel is launched by CPU, the parameters, such as the number of threads per block and the number of blocks per grid, should be specified.

III. Related Work

The BLAST tools are compute- and data-intensive applications, many studies have been proposed to parallelize them on different parallel architectures. NCBI BLAST+ [5] has been proposed to use Pthreads to speedup BLAST on multicore CPU by National Center for Biotechnology Information (NCBI). On CPU clusters, several parallel designs, including TurboBLAST [4], ScalaBLAST [13] and mpiBLAST [2], have been proposed. Among them, mpiBLAST is a widely used design based on NCBI BLAST library. With the efficient task scheduling mechanism and scalable I/O subsystem, mpiBLAST can leverage tens of thousands processors to speedup BLAST. To seek higher throughput, BLAST has also been implemented on various accelerators, such as FPGAs [8], [11], [10]. Mahram et al. [10] have proposed a co-processing approach that leverages both CPU and FPGA to accelerate BLAST. In their design, FPGAs are used to pre-filter dissimilar subject sequences; and then the filtered databases are searched by NCBI BLAST on the CPU.

GPUs are also used to accelerate BLAST. Four GPU BLAST designs, including CUDA-NCBI-BLAST [9], GPU-BLAST [15], CUDA-BLASTP [16] and GPU-BLASTP [17], have been proposed since 2010. CUDA-NCBI-BLAST was the first GPU-implemented BLAST based on NCBI BLAST. The first three steps, including hit detection, ungapped extension and gapped extension, were ported on GPU. However, without GPU architecture-aware optimizations, this implementation achieved only 1.7x to 2.7x speedup on NVIDIA G80 GPU over a single-core Pentium4 CPU. Shortly after it, GPU-NCBI-BLAST built on the NCBI BLAST was proposed. The most time-consuming stages, including the hit detection and ungapped extension, were ported on GPU. With the same accuracy results as NCBI BLAST, the authors claimed approximate four-fold speedup using NVIDIA Fermi GPU over a single-threaded CPU implementation, while two-fold speedup over a multi-threaded CPU implementation on a hexa-core processor. CUDA-BLASTP was proposed to use a compressed DFA for the hit detection with an additional step to sort the sequences in the databases to improve the load balance. CUDA-BLASTP also ported the gapped extension on GPU. GPU-BLASTP improved the load balance further via the runtime work queue design, with which a thread could grab next sequence after completed processing current subject sequence. GPU-BLASTP also provided a two-level buffering for the output of the ungapped extension, where the output

sizes of different sequences could be different since the number of output alignments cannot be obtained in advance. Based on the survey paper [7], GPU-BLASTP is the fastest GPU implementation to date with two-fold speedup over CUDA-BLASTP.

While these designs on GPU illustrated significant speedup over the CPU version, the massively parallel capability of GPU was constrained by the coarse-grained parallelism, where one sequence alignment was mapped to only one thread and different stages of alignment were executed sequentially by this thread. Although some of these studies realized the irregular memory access challenge mapping multiple threads for one sequence alignment, without techniques to eliminate the branch divergence and irregular memory access, the existing research couldn't provide an efficient and fine-grained BLAST design on GPU.

IV. Design of Fine-grained BLASTP

A. Challenges of Mapping BLASTP to GPUs

Figure 3 illustrates the hit detection and ungapped extension phases in the BLASTP algorithm. In the hit detection, each subject sequence in the database will be scanned from left to right, and each word will be compared with all words in the query sequence. All similar words will be tagged as hits. The hit detection is in the column-major order in nature, that means all hits in one column will be tagged at the same time. The ungapped extension is in the diagonal-major order, where two or more hits in one diagonal will be checked to trigger the extension along the diagonal until the gap is encountered or the diagonal is ended.

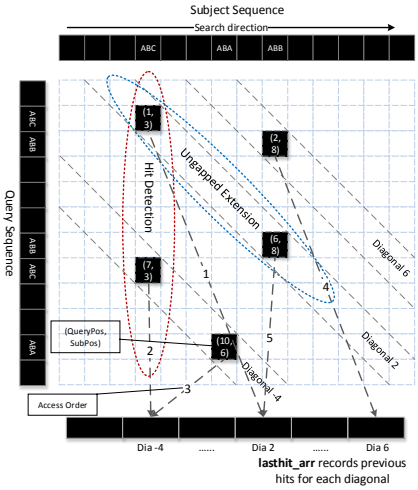


Figure 3: BLASTP Hit Detection and Ungapped Extension

Algorithm 1 illustrates the algorithm used in existing BLASTP research on CPU and GPU. When one hit is detected, the corresponding diagonal number will be calculated as the difference of *hit.sub_pos* and *hit.query_pos* as shown in Line 5. The previous hit in this diagonal will be obtained from the *lasthit_arr* array. If the distance between current hit and previous hit is shorter than a threshold, the ungapped extension is triggered. After finish the extension in current column, the algorithm will move to the next word in the subject sequence.

Algorithm 1 Hit Detection and Ungapped Extension

Input: $subject\_seqs$ subject sequences from the database
Output: $extensions$ extensions for gapped extension

```
1:  for all seq_i in subject_seqs do
2:      for all word_j in seq_i do
3:          hit ← dfa_search(word_j)
4:          if hit ≠ null then
5:              diagonal ← hit.sub_pos − hit.query_pos
                           +query_length
6:              lasthit ← lasthit_arr[diagonal]
7:              disance ← hit.sub_pos − lasthit.sub_pos
8:              if distance < threshold then
9:                  ext ← ungapped_ext(hit, lasthit)
10:                 extensions.add(ext)
11:                 lasthit_arr[diagonal] ← ext.sub_pos
12:             else
13:                 lasthit_arr[diagonal] ← hit.sub_pos
14:             end if
15:         end if
16:     end for
17: end for
```

This algorithm illustrates the interleaving execution of the hit detection and ungapped extension. Due to the heuristic nature, there exists irregular execution paths for different words in the subject sequence. Since the number of hits being able to trigger ungapped extension in different columns cannot be obtained in advance, it is hard to avoid the divergence branch when threads in a same warp are mapped to handle different sequence alignment. Another issue is the random memory access mode in one thread, since the current hit and the previous hit could be in any place of the diagonal. When threads of a warp are used for different sequence alignment, it is hard to organize the coalesced memory access, since each thread has its own previous hit array.

A "fine-grained" multithreaded mode that uses multiple threads unfolding the "for" loop could also lead to severe branch divergence on GPU, considering the uncertain hit number for different words and the uncertain distance with previous hits in diagonals. Furthermore, since any element in the previous hit array may be accessed in any iteration, the "fine-grained" mode may also lead to significant memory access conflict in the irregular mode. Due to these challenges, it is not straightforward to design a fine-grained BLASTP to fully utilize the capability of GPU. We decouple the stages of the BLASTP algorithm and use different strategies to optimize each of them.

B. Hit Detection with Binning

In the fine-grained hit detection, we use multiple threads to detect hit positions in current subject sequence. We issue multiple threads in the column-major order, meaning that the successive threads will handle the consecutive words in the subject sequence. In this way, the memory access for words is efficient in the coalesced pattern. Since the ungapped extension has to be executed along diagonals, the output results of hit detection need to be reorganized by the diagonals. As a result, we introduce the bin data structure and bin-based algorithms for hit detection and ungapped extension.

We first allocate the consecutive buffer in the global memory, and organize the buffer as bins to hold the hit positions. Although one bin can be allocated for one diagonal, considering the increasing length of sequence in databases, we allocate one bin for multiple diagonals to reduce memory usage.

Figure 4: Hit Detection and Binning

Figure 4 illustrates the process of fine-grained hit detection, where one word comparison between the subject sequence and the query sequence will be scheduled to one thread. A thread will get the word from the corresponding position (column number) in the subject sequence, search the word in the DFA to get hit positions (row numbers), and immediately calculate the corresponding diagonal numbers as the difference of row numbers and column number. For example, the thread 3 should get the word "ABC" from the column 3 of the subject sequence, search "ABC" in the DFA to get the hit positions 1, 7, and 11, and calculate the diagonal numbers as 2, -4, and -8. Since different threads can write hit positions into a same bin simultaneously, we have to use atomic operations to write hit positions.

Algorithm 2 Warp-based Hit Detection

Input: $subject\_seqs$ subject sequences from the database
Output: bin bins contains hits

```
1:  tid ← blockDim.x * blockIdx.x + threadIdx.x
2:  numWarps ← gridDim.x * blockDim.x/warpSize
3:  warpId ← tid/warpSize
4:  laneId ← threadIdx.x mod warpSize
5:  i ← warpId
6:  while i < num_seqs do
7:      j ← laneId
8:      while j < seq[i].length do
9:          hits ← dfa_search(seq[i][j])
10:         while hits ≠ null do
11:             hit ← hits.current
12:             diagonal ← hit.sub_pos − hit.query_pos
                           +query_length
13:             binId ← diagonal mod num_bins
14:             curr ← atomicAdd(top[bin_id], 1)
15:             bin[binId][curr] ← hit
16:             hits ← hits.next
17:         end while
18:         j ← j + warpSize
19:     end while
20:     i ← i + numWarps
21: end while
```

Algorithm 2 describes the fine-grained hit detection algorithm. The variable *num_bins* represents the number of bins, which is a configurable parameter in our fine-grained BLASTP algorithm. We set the number of bins to 128 in our experimental evaluation. This algorithm will schedule a warp of threads for a specific sequence. Each word in current thread *seq[i][j]* is handled by the thread with *laneId* *j*. For each hit of the word, the diagonal number will be calculated and mapped to the bin at Line 13. An array named *top* in shared memory is allocated. Each element of this array is used to store current available position in corresponding bin. Using atomic operation on *top*, we can avoid the heavy overhead of atomic operation directly on bins that are allocated on global memory. A warp of threads will be scheduled to next sequence until all words in current sequence are detected.

C. Hit Reorganization with Sorting and Filtering

After the hit detection, hits are grouped into bins by diagonal numbers. Since multiple threads could write hits for different diagonals into a same bin simultaneously, the sequence of hits in each bin are out of order. Because the ungapped extension will determine whether continue the extension based on the distance of two or more neighboring

hits in a diagonal, we have to reorganize hits in each bin to avoid irregular memory access.

Figure 5: Sorting and Filtering with Bin Data Structure

The hit reorganization is to sort the hits by diagonal number with a top-left to bottom-right order. Since one hit is related with row number, column number, diagonal number, and sequence number, we design the bin data structure for the hit to unify the information. As shown in Figure 5, we pack sequence number, diagonal number, and subject position (column number) into a 64 bit length integer. Because more than 99.95% sequences in the most recent NCBI NR database are shorted than 4K letters and the longest sequence contains 36,805 letters, it is enough to use the 16 bit length for the subject position, which can represent 128K positions. Using this packed data structure, we can sort hits in each bin once rather than sort hits twice by diagonal number and subject position, respectively. Another benefit to use this data structure is that when we do the ungapped extension, which needs sequence number, query position (row number) and subject position (column number), the query position can be easily calculated as $subject\_position - diagonal\_number$, and the sequence number can be obtained with the shift operation. With the sorted hits using the specific data structure, the irregular memory access in ungapped extension can be reduced significantly.

After we finish sorting the hits in bins, we add the filtering step to eliminate hits whose distances with neighbors are longer than a threshold, that means these hits can not be used to trigger the ungapped extension (based on two or more hits in each diagonal). A warp of threads are used to eliminate hits for each sequence in one bin. A thread scheduled for one hit compares the threshold with the distance to the neighbor on left and then the distance to the neighbor on right. Only when the distances to two neighbors are longer than the threshold, the hits will be eliminated. The overall performance with the additional filtering step is determined by the ratio of overhead of hit filtering and the overhead of the branch divergence. For the datasets used in our experimental evaluation, we have observed only 5% to 11% hits from the hit detection stage can be used to trigger the ungapped extension. As a result, the overall performance is improved with the hit filtering.

D. Fine-grained Ungapped Extension

After the hit reorganization with sorting and filtering, the hits in each bin are arranged in ascending order by diagonals and the hits whose distances with neighbors are longer than the threshold are eliminated. Based on the ordered hits, we design a diagonal-based ungapped extension algorithm that is illustrated in Algorithm 3, where one diagonal will be scheduled to one thread for the ungapped extension. As shown from Line 6 to Line 8, different thread warps are scheduled to different bins and threads in a warp are scheduled to different diagonals. We get the sequence number *seq_id*, the column number *sub_pos*, and the row number *query_pos* from the bin data structure and call *ungapped_ext* function to extend the diagonal until a gap is encountered or the diagonal is ended. The variable *ext* represents the extension result. Since an extension could cover other hits along the diagonal, Line 17 is used to determine whether a hit is covered by the previous extension. Only if a hit is not covered by the previous extension, we trigger the extension from this hit.

Algorithm 3 Diagonal-based Ungapped Extension

Input: bin binned hits
Output: $extensions$ extensions for gapped extension

```
1:  tid ← blockDim.x * blockIdx.x + threadIdx.x
2:  numWarps ← gridDim.x * blockDim.x/warpSize
3:  waprId ← tid/warpSize
4:  laneId ← threadIdx.x mod warpSize
5:  i ← warpId
6:  while i < num_bins do
7:      j ← laneId
8:      while j < bin_i.num_diagonals do
9:          dia_start ← dia_offset[j]
10:         dia_end ← dia_offset[j + 1]
11:         ext_reach ← −1
12:         for k ← dia_start to dia_end do
13:             hit ← bin[i][k]
14:             sub_pos ← hit.sub_pos
15:             query_pos ← hit.sub_pos
                            −hit.diag_num
16:             seq_id ← hit.seq_id
17:             if sub_pos > ext_reach then
18:                 ext ← ungapped_ext(seq_id,
                                query_pos, sub_pos)
19:                 extensions.add(ext)
20:                 ext_reach ← ext.sub_pos
21:             end if
22:         end for
23:         j ← j + warpSize
24:     end while
25:     i ← i + numWarps
26: end while
```

Since there are still divergence branches in the diagonal-

Algorithm 4 Hit-based Ungapped Extension

Input: bin binned hits
Output: $extensions$ extensions for gapped extension

```
1: tid ← blockDim.x * blockIdx.x
         +threadIdx.x
2: numWarps ← gridDim.x * blockDim.x/warpSize
3: waprId ← tid/warpSize
4: laneId ← threadIdx.x mod warpSize
5: i ← warpId
6: while i < num_bins do
7:     j ← laneId
8:     while j < bin_i.num_hits do
9:         j ← j + warpSize
10:        hit ← bin[i][j]
11:        sub_pos ← hit.sub_pos
12:        query_pos ← hit.sub_pos − hit.diag_num
13:        seq_id ← hit.seq_id
14:        ext ← ungapped_ext(seq_id,
                      query_pos, sub_pos)
15:        extensions.add(ext)
16:        j ← j + warpSize
17:    end while
18:    i ← i + numWarps
19: end while
```

based extension algorithm, we design a hit-based ungapped extension to eliminate the divergence further. Algorithm 4 illustrates the hit-based ungapped extension. We schedule one thread to one hit and start the extension per hit independently. Since the extension results from different hits could be the same, the hit-based extension may write duplication at Line 15 with the redundant computation. We leave the result de-duplication in the following stag running on CPU. The performance comparison between the hit-based ungapped extension and the diagonal-based ungapped extension depends on the characters of the sequence. If there are too many hits that can be covered by the extension from other hits in diagonals, the diagonal-based ungapped extension should performs better. As a result, we use a configurable parameter to allow the user to select the ungapped extension algorithms at runtime.

(a) Coarse-grained (b) Diagonal-based (c) Hit-based

Figure 6: Three Level of Parallelism of Ungapped Extension

Figure 6 compares the parallelism mode of different ungapped extension algorithms. Figure 6(a) illustrates the coarse-grained ungapped extension in existing research. Since the hit detection and ungapped extension are interleaved, the coarse-grained ungapped extension extends hits from different diagonals in a sequence sequentially. Our warp-based algorithms can extend hits in a sequence in parallel.

E. Hierarchical Buffering

To fully utilize memory bandwidth, we propose a hierarchical buffering for the core data structure DFA used in the hit detection. As shown in Figure 2(a), DFA consists of the states in the finite state machine and the query positions for the states. Both the states and query positions are highly reused in hit detection for each word. Loading DFA into shared memory can improve the data access bandwidth. However, because the number of query positions depend on the query length, prefetching all positions into the shared memory may affect the occupancy of GPU kernels and offset the improvement from higher data access bandwidth, especially for the long sequences. Thus, we load the states having relatively fixed and small sizes into the shared memory, and store query positions on the constant memory. On the latest NVIDIA Kepler GPU, a 48KB read-only cache with relaxed memory coalescing rules is introduced for the reusable but randomly accessed data. We allocate the query positions in the global memory and tag them with the keyword "const __restrict" for loading them into the read-only cache automatically.

Figure 7: Hierarchical Buffering for DFA on Kepler GPU

Figure 7 illustrates the hierarchical buffering architecture for DFA on Kepler GPU. We put the DFA states, e.g., "ABB" and "ABC", into the shared memory. For the first access of "ABC" from thread 3, the positions are set into bins and loaded into the read-only cache. For the following access of "ABC" from thread 13, the positions will be obtained from the cache.

PSS matrix (or scoring matrix) is core data structure highly reused in the ungapped extension. The number of column in PSS matrix is equal to the length of the query sequence. Since each column contains 64 Bytes (32 rows with 2 Bytes for each), the size of PSS matrix increase quickly with the query length. The 48KB shared memory cannot hold the PSS matrix for the query sequence longer than 768. Furthermore, too many shared memory usage will degrade the performance due to the degraded GPU occupancy. On the other hand, if the scoring matrix is used to substitute the PSS matrix, the scoring matrix with the fixed 2KB size can be always fit into the shared memory, while more memory accesses using scoring matrix could decrease

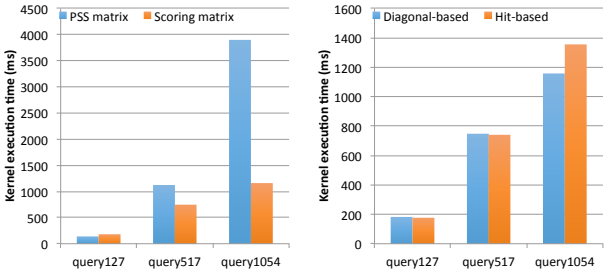
the performance compared with using PSS matrix for short sequences. Thus, we provide a configurable parameter to select PSS matrix or scoring matrix. For the PSS matrix, we put it into the shared memory until a threshold and then we put it into the global memory. For the scoring matrix, we always put it into the shared memory. We will compare the performance using the PSS matrix and the scoring matrix in Section V.

V. Performance Evaluation

We carry out our experimental evaluation on two compute nodes. Each node has Intel Core i5-2400 quad-core processor with 6MB shared L3 cache, and 8GB DDR3 main memory. One node is installed with NVIDIA Fermi GPU, and the other one is installed with NVIDIA Kepler K20c GPU. Each node has Debian Linux 3.2.35-2 and NVIDIA CUDA toolkit 5.0. We use two typical databases from NCBI. One is *env_nr* database including 6 million sequences with the total size at 1.7 GB. The average sequence length is around 200 letters. Another one is *swissprot* database including 300,000 sequences with the total size at 150 MB. The average sequence length is around 370 letters. For each database, we choose three sequences as the query sequences, which lengths are 127 Byte, 517 Byte, and 1054 Byte, respectively. They are represented as "query127", "query517", and "query1054" in the following figures.

A. Configurable Parameter Evaluation

Since the configurable parameters are more sensitive to the query sequence instead of the subject sequence in the database, we use database *env_nr* to investigate the relationship between the performance and different configurations. We first compare the performance using the PSS matrix and the scoring matrix. Figure 8(a) illustrates their performance comparison on NVIDIA Kepler K20c GPU. Compared with PSS matrix, we observe -24%, 50%, and 237% performance improvement using scoring matrix for "query127", "query517", and "query1054", respectively. As mentioned in Section IV-E, PSS matrix size is larger than the shared memory size when the length of the query sequence is longer than 768 Byte. For the sequence longer than that, we have to move PSS matrix to the constant memory. Using scoring matrix for the medium sequence and long sequence (i.e., "query517" and "query1054"), we can reduce the shared memory usage, in turn improve the occupancy of GPU kernel. For the short sequence "query127", the PSS matrix can be fit into the shared memory. Using the PSS matrix, we can decrease the memory access operations compared with using the scoring matrix. As a result, we configure our algorithm to use the PSS matrix for "query127" and the scoring matrix for "query517" and "query1057" on NVIDIA Kepler K20c GPU for the following evaluations. Since we observe the similar performance comparison on NVIDIA Fermi GPU, we omit the performance comparison on Fermi for saving space.



(a) PSS Matrix vs. Scoring Matrix (b) Hit-based vs. Diagonal-based

Figure 8: Performance Comparison with Different Configurations

We also compare the performance using the diagonal-based ungapped extension and the hit-based ungapped extension on NVIDIA Kepler K20c GPU. As illustrated in Figure 8(b), compared with the hit-based extension, the diagonal-based extension gets -3%, 1%, and 21% performance improvement for "query127", "query517", and "query1054", respectively. Because the diagonal length significantly affects the execution time of ungapped extension and the diagonal length is only determined by the query sequence length, less redundant computation is observed in the hit-based extension in "query127" than "query517" and "query1054". We configure our algorithm to use the hit-based extension for "query127" and the diagonal-based extension for other two on NVIDIA Kepler K20c GPU in the following evaluation. We observe the similar tendency on NVIDIA Fermi GPU and then set the same configurations in our algorithm.

B. Performance Comparison with Existing BLASTP Algorithms

Figure 9 illustrates the normalized speedup of our cuBLASTP running on NVIDIA Fermi GPU and Kepler GPU over the sequential FSA-BLAST on CPU, the multithreaded NCBI-BLAST on CPU, and GPU-BLASTP on GPUs, which is the fastest BLAST on GPU to date.

Compared with FSA-BLAST, Figure 9(a) and Figure 9(b) illustrate on the database *env_nr*, cuBLASTP has up to 7.8-fold speedup and 4.7-fold speedup for the critical phases, including the hit detection and ungapped extension, and the overall performance, respectively. Figure 9(c) and Figure 9(d) show that on the database *swissprot*, cuBLASTP has up to 5.3-fold speedup and 5-fold speedup for the critical phases and the overall performance, respectively.

Compared with NCBI-BLAST with 4 threads, Figure 9(e) and Figure 9(f) illustrate on the database *env_nr*, cuBLASTP has up to 2.9-fold speedup and 1.8-fold speedup for the critical phases and the overall performance, respectively. Figure 9(g) and Figure 9(h) show that on the database *swissprot*, cuBLASTP has up to 2.4-fold speedup and 3.7-fold speedup for the critical phases and the overall performance, respectively.

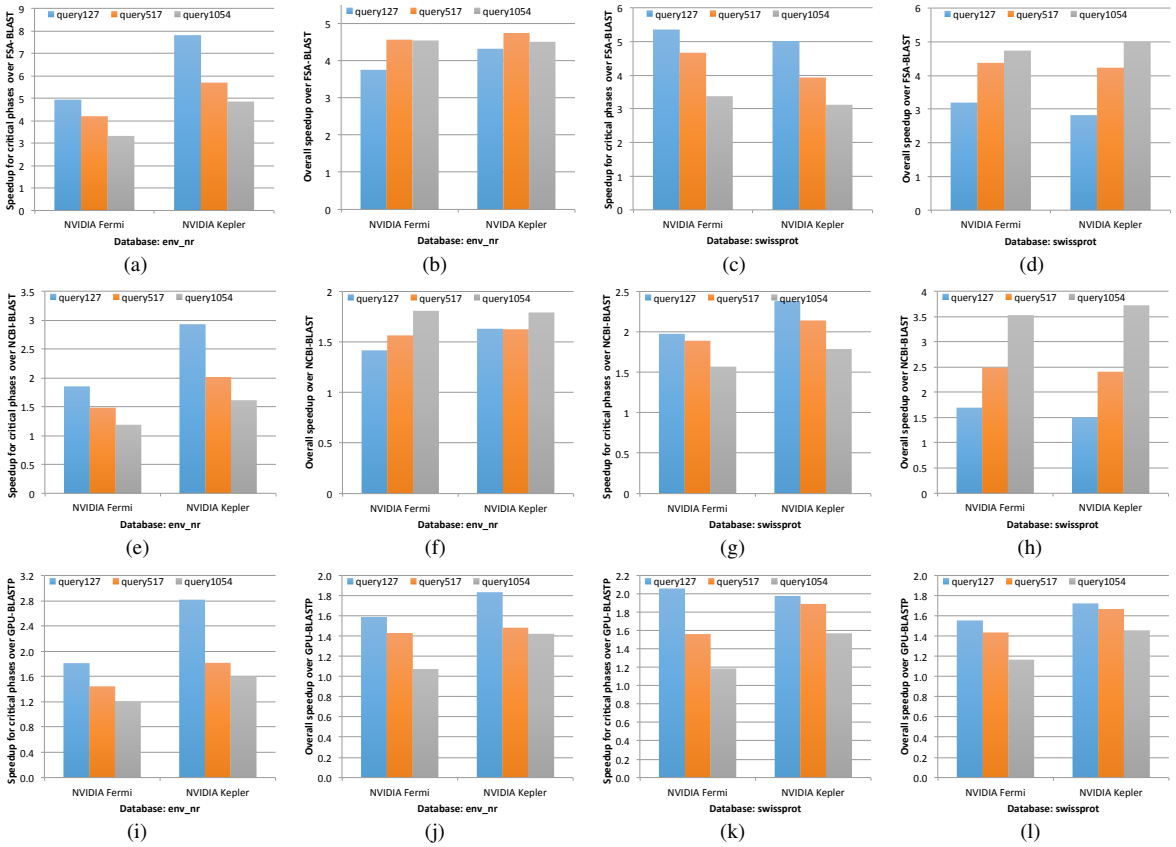


Figure 9: Speedup of cuBLASTP over sequential FSA-BLASTP(a-d), NCBI-BLAST with four threads(e-h) and GPU-BLASTP(i-l)

Compared with GPU-BLASTP, Figure 9(i) and Figure 9(j) illustrate on the database *env_nr*, cuBLASTP has up to 2.8-fold speedup and 1.8-fold speedup for the critical phases and the overall performance, respectively. Figure 9(k) and Figure 9(l) show that on the database *swissprot*, cuBLASTP has up to 2.1-fold speedup and 1.7-fold speedup for the critical phases and the overall performance, respectively.

Our cuBLASTP has overlapped data transfer between CPU and GPU with the kernel execution to improve the overall performance. As a result, cuBLASTP has shown the consistent improvement of the overall performance. Figure 9(i)- 9(l) illustrate cuBLASTP has better performance on NVIDIA Kepler than that on Fermi. It is due to the optimization related with the Kepler architecture, such as the hierarchical buffer using the read-only cache.

Figure 10(a), Figure 10(b), and Figure 10(c) show the profiling results of GPU-BLASTP and cuBLASTP on NVIDIA Kepler K20c GPU using NVIDIA Visual Profiling tool. For the limited space, we only show the results for "query517" on *env_nr* database. Since the GPU-BLASTP has the interleaved execution mode, we get the profiling number for one fused kernel; and for cuBLASTP, we can get the profiling results for separate kernels of hit detection, hit sorting and filtering, and ungapped extension, respectively.

Figure 10(a) illustrates 67.0%, 46.2%, 6.1% global memory load efficiency for three cuBLASTP kernels, and 11.5% for GPU-BLASTP fused kernel. The good efficiency in our hit detection, and sorting and filtering comes from the coalesced memory access mode: in the hit detection, the threads in a same warp access positions of the subject sequence successively; and in the sorting and filtering, the threads in a same warp also access hits in one bin successively. However, the load efficiency in our ungapped extension is low. The ungapped extension needs to load next position along the diagonal to check whether the extension can be continued. Thus, any position could be loaded from the subject sequence by different threads. Due to the random access mode, we didn't guarantee to load positions successively by threads in a same warp. Figure 10(d) illustrates the execution time breakdown of cuBLASTP for "query517". The time spending in the ungapped extension stage is 13% of the total execution time. As a result, although the load efficiency of the ungapped extension is only 6.1%, it doesn't affect our overall speedup. Specifically, as shown in Figure 10(b) and Figure 10(c), all three kernels of cuBLASTP have lower branch divergence overhead and higher GPU occupancy achieved than the GPU-BLASTP fused kernel. The profiling number provides the insight to explain why cuBLASTP can perform better than GPU-BLASTP. Finally, we would like to mention our accuracy number. The outputs of cuBLASTP are totally identical to the outputs of FAS-BLAST on CPU. It illustrates the accuracy of our implementation is same as

FSA-BLAST.

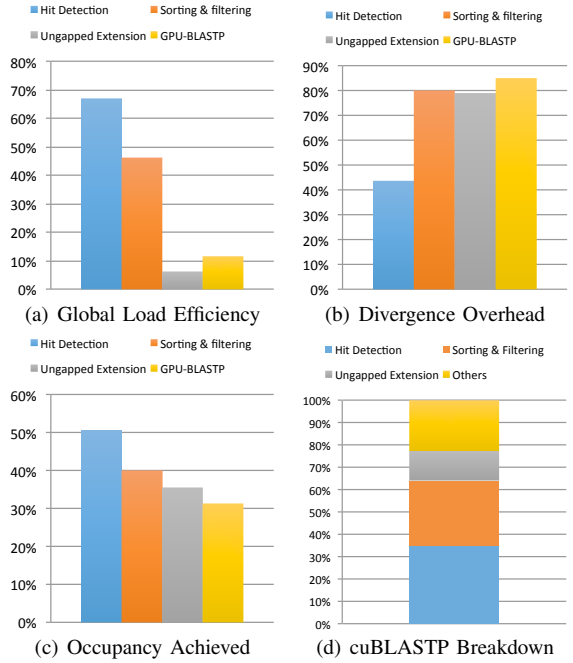


Figure 10: Profiling on cuBLASTP and GPU-BLASTP

VI. Conclusion and Future Work

In this paper, we propose cuBLASTP, an efficient fine-grained BLASTP for GPU using the CUDA programming model. We decompose the hit detection and ungapped extension into separate phases and use different GPU kernels to speedup their performance. In order to eliminate the branch divergence and irregular memory access, we propose the bin data structure based optimizations, including the memory access reordering, the position-based indexing, the hit sorting and filtering, and the diagonal-based and the hit-based parallelisms for ungapped extension. We also propose a hierarchical buffering for the core data structures to take advantage of the latest NVIDIA Kepler architecture. On NVIDIA Kepler GPU, cuBLASTP has up to 7.8-fold speedup over FSA-BLAST on a single core and 2.9-fold speedup over NCBI-BLAST on a quad-core CPU for the critical phases and up to 5-fold and 3.7-fold speedup and for the overall performance. Compared with GPU-BLASTP, the existing fastest GPU BLAST algorithm, cuBLASTP has up to 2.8-fold speedup for the critical phases and up to 1.8-fold speedup for the overall performance.

In the future, we would like to extend cuBLASTP on large scale systems with GPUs to match the requirement of next generation sequencing in big data era.

Acknowledgment

This research was supported in part by NSF via IIS-1247693.

References

[1] FSA-BLAST. http://sourceforge.net/projects/fsa-blast/.

[2] Aaron E. Darling and Lucas Carey and Wu-chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *ClusterWorld Conference & Expo 2003*, 2003.

[3] Altschul, S. F. and Gish, W. and Miller, W. and Myers, E. W. and Lipman, D. J. Basic Local Alignment Search Tool. *Journal of molecular biology*, 215(3):403–410, Oct. 1990.

[4] Bjornson, R. D. and Sherman, A. H. and Weston, S. B. and Willard, N. and Wing, J. TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub. In *IPDPS '02*, 2002.

[5] Camacho, Christiam and Coulouris, George and Avagyan, Vahram and Ma, Ning and Papadopoulos, Jason S. and Bealer, Kevin and Madden, Thomas L. BLAST+: Architecture and Applications. *BMC Bioinformatics*, 10:421, 2009.

[6] Cameron, Michael and Williams, Hugh E. and Cannane, Adam. A Deterministic Finite Automaton for Faster Protein Hit Detection in BLAST. *Journal of Computational Biology*, 13(4):965–978, 2006.

[7] David Glasco. An Analysis of BLASTP Implementation on NVIDIA GPUs, 2012.

[8] Jacob, Arpith C. and Lancaster, Joseph M. and Buhler, Jeremy and Harris, Brandon and Chamberlain, Roger D. Mercury BLASTP: Accelerating Protein Sequence Alignment. *TRETS*, 1(2), 2008.

[9] Ling, Cheng and Benkrid, Khaled. Design and Implementation of a CUDA-compatible GPU-based Core for Gapped BLAST Algorithm. In *ICCS '10*, 2010.

[10] Mahram, Atabak and Herbordt, Martin C. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-based Prefiltering. In *ICS '10*, 2010.

[11] Muriki, Krishna and Underwood, Keith D. and Sass, Ron. RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation. In *IPDPS '05*, 2005.

[12] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2010. Version 3.2.

[13] Oehmen, Chris and Nieplocha, Jarek. ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Trans. Parallel Distrib. Syst.*, 17(8):740–749, 2006.

[14] Smith, T. and Waterman, M. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1): 195–197, 1981.

[15] Vouzis, Panagiotis D. and Sahinidis, Nikolaos V. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. *Bioinformatics*, 27(2):182–188, 2011.

[16] Weiguo Liu and Bertil Schmidt and Wolfgang Muller-Wittig. CUDA-BLASTP: Accelerating BLASTP on CUDA-enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*.

[17] Xiao, Shucai and Lin, Heshan and Feng, Wu-chun. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *IPDPS '11*, 2011.