This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

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

Jing Zhang, Hao Wang, Wu-chun Feng

Abstract—BLAST, short for Basic Local Alignment Search Tool, is a ubiquitous tool used in the life sciences for pairwise sequence search. However, with the advent of next-generation sequencing (NGS), whether at the outset or downstream from NGS, the exponential growth of sequence databases is outstripping our ability to analyze the data. While recent studies have utilized the graphics processing unit (GPU) to speedup the BLAST algorithm for searching protein sequences (i.e., BLASTP), these studies use coarse-grained parallelism, where one sequence alignment is mapped to only one thread. Such an approach does not efficiently utilize the capabilities of a GPU, particularly due to the irregularity of BLASTP in both execution paths and memory-access patterns. To address the above shortcomings, we present a fine-grained approach to parallelize BLASTP, where each individual phase of sequence search is mapped to many threads on a GPU. This approach, which we refer to as cuBLASTP, reorders data-access patterns and reduces divergent branches of the most time-consuming phases (i.e., hit detection and ungapped extension). In addition, cuBLASTP optimizes the remaining phases (i.e., gapped extension and alignment with trace back) on a multicore CPU and overlaps their execution with the phases running on the GPU.

Index Terms—BLAST, BLASTP, GPU, bioinformatics, algorithmic refactoring, fine-grained parallelization, life sciences, local alignment, pairwise sequence search, next-generation sequencing.

1 Introduction

The *Basic Local Alignment Search Tool (BLAST)* [25] is a fundamental algorithm in the life sciences that compares a query sequence to database of known sequences in order to identify the most similar known sequences to the query sequence. The similarities identified by BLAST can then be used to infer functional and structural relationships between the corresponding biological entities, for example.

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 to accelerate sequence-alignment tools, such as BLAST, on various parallel architectures in recent years.

Graphics processing units (GPUs) offer the promise of accelerating bioinformatics algorithms and tools due to their superior performance and energy efficiency. However, in spite of the promising speedups that have been reported for other sequence alignment tools such as Smith-Waterman [27], BLAST remains the most popular sequence analysis tool but also one of the most challenging to accelerate on GPUs.

Due to its popularity, the BLAST algorithm has been heavily optimized for CPU architectures over the past two decades. However, these CPU-oriented optimizations create problems when accelerating BLAST on GPU architectures. First, to improve computational efficiency, BLAST employs input-sensitive heuristics to quickly eliminate unnecessary search spaces. While this technique is highly effective on CPUs, it induces unpredictable execution paths in the program, leading to many divergent branches on GPUs. Second, to improve memory-access efficiency, the data structures used in BLAST are finely tuned to leverage CPU caching. Re-using these data structures on GPUs, however, can lead to highly inefficient memory access because the cache size on GPUs is significantly smaller than that on CPUs and because coalesced memory access is needed on GPUs to achieve good performance.

State-of-the-art BLAST realizations for protein sequence search on GPUs [29], [26], [23], [9] adopt a coarse-grained and embarrassingly parallel approach, where one sequence alignment is mapped 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 GPUs. However, such an approach presents significant challenges, mainly due to the high irregularity in execution paths and memory-access patterns that are found in CPU-based realizations of the BLAST algorithm. Thus, accelerating BLAST on GPUs requires a fundamental rethinking in the algorithmic design of BLAST.

Consequently, we propose cuBLASTP, a novel fine-grained mapping of the BLAST algorithm for protein search (BLASTP) onto a GPU, that improves performance by addressing the irregular execution paths caused by branch divergence and irregular memory access. First, we decouple the phases in the BLASTP

• *J. Zhang, H. Wang, and W. Feng are with the Dept. of Computer Science at Virginia Tech. E-mail: {zjing14, hwang121, wfeng}@vt.edu.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

algorithm (i.e., hit detection, ungapped extension, gapped extension, and alignment with traceback) and parallelize the phases having different computational patterns with different strategies on the GPU or CPU, as appropriate. Second, we propose a binning-sorting-filtering optimization as an additional phase between the phases of BLASTP to reduce branch divergence and irregular memory access. This optimization includes the following: (a) reordering the memory-access pattern from column-major order in the hit-detection phase to diagonal-major order, as in the ungapped-extension phase; and (b) reducing divergence branches in the ungapped-extension phase by sorting hits in each diagonal and eliminating hits beyond the threshold distance along the diagonal. Third, we propose diagonal-based parallelism, hit-based parallelism, and window-based parallelism for the ungapped-extension phase to efficiently extend sequences with different characteristics in databases. Fourth, we design a hierarchical buffering mechanism for the core data structures, e.g., deterministic finite automation (DFA) and position-specific scoring matrix (PSS matrix), using features provided by the NVIDIA Kepler architecture, e.g., read-only cache, to improve data-access bandwidth on the GPU. Finally, we also optimize the remaining phases of BLASTP, i.e., gapped extension and alignment with traceback, on a multicore CPU and overlap the different phases running on the CPU with those running on the GPU.

2 Background

In this section, we provide a brief discussion of the BLAST algorithm and GPU architecture.

2.1 Basic Local Alignment Search Tool (BLAST)

BLAST is a family of programs with variants used for different types of alignment searches. BLAST algorithms approximate the results of the Smith-Waterman algorithm, an optimal local-alignment algorithm. Instead of comparing entire sequences, BLAST algorithms locate high-scoring short matches (i.e., hits) between a query sequence and subject sequences and extend the hits to longer alignments. With only a slight loss in accuracy, BLAST algorithms execute significantly faster than Smith-Waterman. In this paper, we focus on BLASTP, which compares the similarity of protein sequences.

We use FSA-BLAST [2], which has been optimized on the CPU for protein sequence search, as an example to illustrate the BLAST algorithm. The BLAST algorithm consists of four phases, as described below.

Hit detection finds high-scoring short matches (i.e., hits) between the query sequence and each subject sequence from the collection of sequences being searched (i.e., sequence database). The short matches are subsequences (i.e., words) with fixed length W (typically, $W = 3$ for protein sequence search; $W = 11$ for nucleotide sequence search) extracted from the query and the subject sequence. To quickly detect hits, the query sequence is preprocessed and converted into a lookup table. Alternatively, for nucleotide sequence search, instead of preprocessing a query sequence into a lookup structure, the lookup table can be built upon the database for better performance (e.g., MegaBLAST [7]). However, there exists *no* BLASTP tool that uses a database index. Why? The index for protein sequence search has a larger alphabet size, short words, and neighboring words, leading to an exponential increase in index size and search complexity. To help address these issues, alternative (non-BLAST) algorithms that make substantial changes in searching methods and scoring mechanisms, e.g., CAFE [13], BLAT [28] and SSAHA [30], have been proposed in order to facilitate an approach based on database indexing. However, while these (non-BLAST) database-index approaches report much better performance than BLAST, they suffer from poor sensitivity and less accuracy than BLAST [18]. Consequently, we focus on the standard BLASTP algorithm rather than those variants using database index.

Ungapped extension determines if two or more hits from hit detection can form the basis of a local alignment without insertions and deletions. It also passes extended hits with the requested scores to the next phase. Only hits along the same diagonal can trigger ungapped extension.

Gapped extension performs a gapped alignment with dynamic programming on high-scoring ungapped regions to determine if they can be part of a larger, higher-scoring alignment.

Alignment with traceback re-scores all the alignments from the gapped extension using a traceback algorithm and produces the top scores.

Fig. 1 illustrates the phases of BLASTP. The word IYP is detected as a hit between the query and the subject sequence in the hit-detection phase. The hit is then extended to a larger ungapped region in the ungapped-extension phase. This region is extended further to a region with insertions and deletions in the gapped-extension phase. Finally, the alignment with traceback re-computes the score of the alignment.

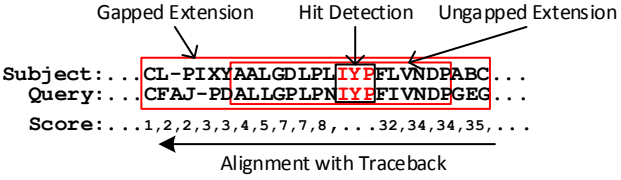


Fig. 1: Four Stages of BLAST Execution [26]

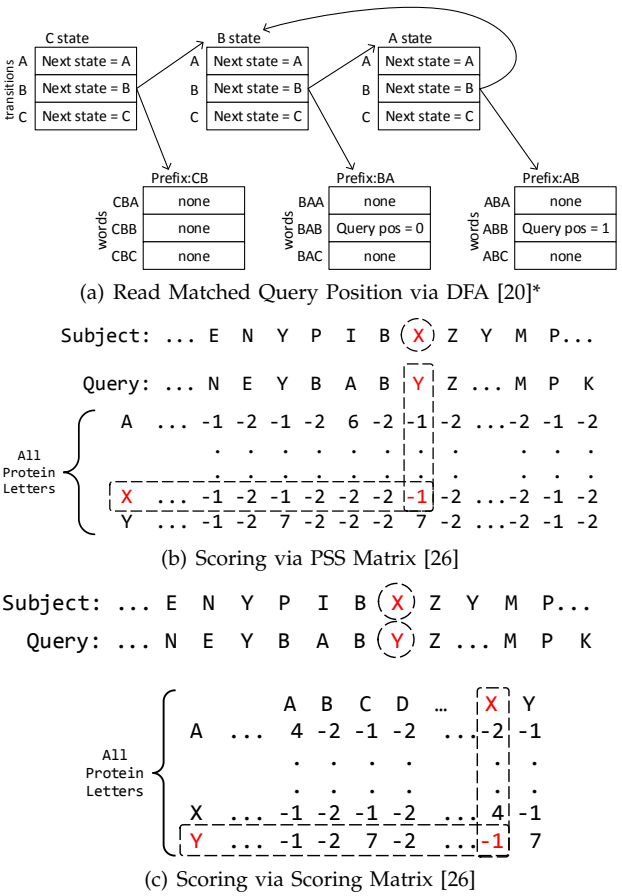
Based on [19], where 100 queries are randomly chosen from the NR protein database [11] and profiled, hit detection and ungapped extension consume the most time, taking nearly 70% of the total execution time. Thus, our work in this paper focuses on the optimizations of these two phases on the GPU.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

Below we describe the core data structures used in hit detection and ungapped extension: deterministic finite automaton (DFA) [20]; position-specific scoring matrix (PSS matrix or PSSM); and scoring matrix. The *DFA* 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 an example, Fig. 2(a) shows the portion of DFA structure that is traversed when the example subject sequence "CBABB" is processed (and when the word length is 3 and query sequence is "BABBC"). First, the letter C is read, and the current state is C. Because the next letter is B, the next state of the DFA transitions to the B state. Simultaneously, the DFA provides a pointer to the CB prefix words to retrieve the query positions for the word CBA. Because the position for CBA in the DFA constructed from $BABBC$ is "none," there is no hit found for CBA. Likewise, for the next letter A in $CBABB$, the DFA transitions to the A state and provides a pointer to the BA prefix words to retrieve the query positions for the word BAB, which is in position 0 in $BABBC$, and so on.

The *PSS matrix* is built from the query sequence. As shown in Fig. 2(b), a column in the PSS matrix represents a position in the query sequence, and the scores in the rows indicate the similarity of all symbols (i.e., amino acid) to the symbol in the column of the query sequence. So, the score for X in the subject sequence and Y in the query sequence is -1. By checking the PSS matrix, the BLAST algorithm can quickly identify the similarity between two symbols in corresponding positions of two sequences.

The *scoring matrix* is an alternative data structure of the PSS matrix. This matrix has a fixed and smaller size than the PSS matrix because the elements in the columns represent words instead of positions in the PSS matrix. The drawback in using this scoring matrix is that more memory accesses are needed. For example, to compare the same pair of letters as above, Fig. 2(c) shows that BLAST must first load the letter X from the subject sequence *and* Y from the query sequence, and then it can retrieve the score of -1 from column X and row Y.



(a) Read Matched Query Position via DFA [20]*

(b) Scoring via PSS Matrix [26]

(c) Scoring via Scoring Matrix [26]

Fig. 2: Core Data Structures in BLAST. In Fig 2(a), $W = 3$ and the example query sequence is $BABBC$, and the example subject sequence is $CBABB$.

2.2 GPU Architecture and Programming Model

A NVIDIA GPU contains a set of streaming multiprocessors (SMs), each consisting of multiple cores of single-instruction, multiple-thread (SIMT) units. There are two types of memory on the GPU: on-chip memory and off-chip memory. On-chip memory, such as the register file and shared memory, has low access latency but relatively small size. Off-chip memory, e.g., global memory, has much larger size but higher access latency. To efficiently access data in off-chip memory, read/write operations must be coalesced. The latest NVIDIA Kepler architecture also offers various caches to improve the efficiency of data access, especially for those with irregular access patterns. Specifically, a 48-kB read-only cache is introduced to improve irregular memory-access performance.

CUDA [22] is a programming model provided by NVIDIA. The CUDA functions that run on a GPU are called kernels. A kernel runs a large number of threads in parallel on the GPU. The threads are grouped into blocks of threads, and in turn, grids of blocks. Thread execution on the GPU follows a SIMT model, where threads running on a SM are partitioned into groups (i.e., warps) and execute the same instruction simultaneously. If the threads in a single warp take different execution paths (i.e., branch divergence), these different paths within a warp are serialized, thus causing lower resource utilization and adversely impacting GPU performance.

3 Design of a Fine-Grained BLASTP

Here we first analyze the challenges in our coarse-grained BLASTP algorithm on the GPU. Then we introduce our fine-grained BLASTP algorithm. The basic idea is to explicitly partition the phases of BLASTP from within a single kernel into multiple kernels, where each kernel is optimized to run across a *group of*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

GPU threads. In particular, this is done for hit detection and ungapped extension. We then present our CPU-based optimizations for the two remaining phases, i.e., gapped extension and alignment with traceback.

3.1 Challenges of Mapping BLASTP to GPUs

Fig. 3 shows how hit detection and ungapped extension execute in the default BLASTP algorithm. In hit detection, each subject sequence in the database is scanned from left to right to generate words; each word in turn is searched in the DFA of the query sequence. The positions with similar words found in the query sequence are tagged as hits, with each hit denoted as a tuple with two elements $(QueryPos, SubPos)$, i.e., the positions in the query sequence and subject sequence, respectively. For example, the word ABC in the subject sequence is searched in the DFA and found in positions 1, 7, and 11 of the query sequence, which in turn generates the following tuple hits: $(1, 3)$, $(7, 3)$, and $(11, 3)$.

After finding the hits, the BLASTP algorithm starts the ungapped extension. The algorithm uses a global array denoted as *lasthit_arr* to record the hits found in the previous detection for each diagonal. In ungapped extension, the algorithm checks the previous hits in the same diagonals with the current hits. If the distance between the previous hit and the current hit is smaller than a threshold, ungapped extension continues until a gap is encountered. For example, when the word ABB is processed to generate the hits $(2, 8)$ and $(6, 8)$, the hits in the *lasthit_arr* array for diagonal 2 and diagonal 6 are checked.

Because all the hits in one column are tagged simultaneously, hit detection proceeds in *column-major order*. However, ungapped extension proceeds in *diagonal-major order*, where hits in a diagonal are checked from the top left to bottom right. Fig. 3 also illustrates the memory-access order on the *lasthit_arr* array. With the interleaved execution of hit detection and ungapped extension, memory access on the *lasthit_arr* array is highly irregular.

Algorithm 1 illustrates the traditional BLASTP algorithm, on either CPU or GPU. When a hit is detected, the corresponding diagonal number is calculated as the difference of *hit.sub_pos* and *hit.query_pos*, as shown in Line 6. The previous hit in this diagonal is obtained from the *lasthit_arr* array (Fig. 3). If the distance between the current hit and previous hit is less than a certain threshold, the ungapped extension is triggered. After ungapped extension occurs in the current diagonal, the extended position in the subject sequence is used to update the previous hit in the *lasthit_arr* array. After all hits in the current column are checked in the ungapped-extension phase, the algorithm moves forward to the next word in the subject sequence.

Fig. 4 shows how the BLASTP algorithm traditionally maps onto a GPU. It is a coarse-grained approach

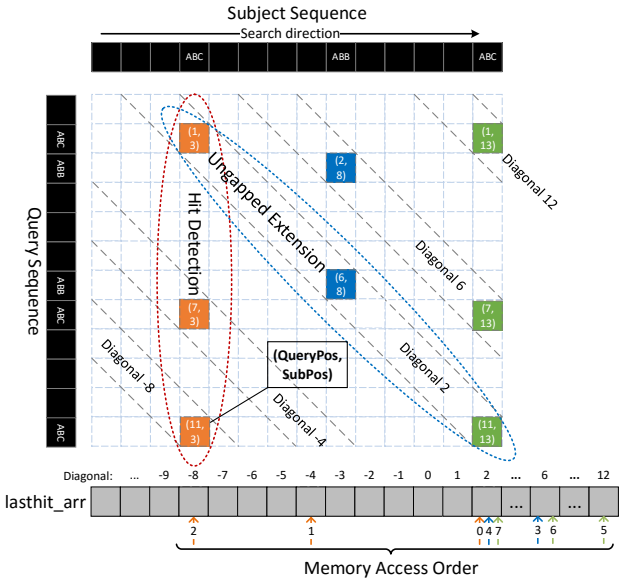


Fig. 3: BLASTP Hit Detection and Ungapped Extension

Algorithm 1 Hit Detection and Ungapped Extension

Input: $database$: sequence database;
DFA: DFA lookup table base on query sequence
Output: $extensions$: results of ungapped extension

```
1:  for all sequence_i in database do
2:      for all word_j in sequence_i do
3:          find hits for word_j in DFA
4:          for all hit_k in hits do
5:              //calculate diagonal number
6:              diagonal ← hit_k.sub_pos − j
                                        + query_length
7:              //get lasthit in the same diagonal
8:              lasthit ← lasthit_arr[diagonal]
9:              //calculate distance to lasthit
10:             distance ← hit_k.sub_pos
                                    − lasthit.sub_pos
11:             if distance within threshold then
12:                 //perform ungapped extension
13:                 ext ← ungapped_ext(hit_k, lasthit)
14:                 extensions.add(ext)
15:                 //update lasthit with ext position
16:                 lasthit_arr[diagonal] ← ext.sub_pos
17:             else
18:                 //update lasthit with hit position
19:                 lasthit_arr[diagonal] ← hit.sub_pos
20:             end if
21:         end for
22:     end for
23: end for
24: output extensions
```

where all the phases of the alignment between the query sequence and one subject sequence are handled by a dedicated thread on the GPU. Because of the heuristic nature of BLASTP, there exist irregular execution paths in different subject sequences from a sequence database. Since the number of hits that trigger ungapped extension in different sequences cannot be deduced in advance, branch divergence (and in turn, load imbalance) occurs when using coarse-grained parallelism in BLASTP. For example, while thread 2 works on ungapped extension, as shown in Fig. 4,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

neither thread 0 nor thread 1 can trigger because in thread 0, there is no hit found in hit detection, and in thread 1, the distance between the current hit and previous hit is larger than the threshold T. As a result, the branch divergence in this warp cripples BLASTP performance on a GPU.

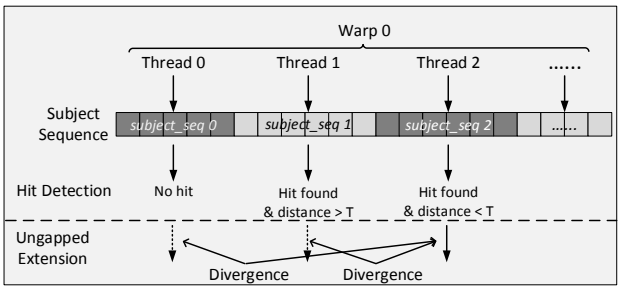


Fig. 4: Branch Divergence in Coarse-Grained BLASTP

Irregular memory access further impacts BLASTP performance on a GPU. Because the current hits can lead to irregular memory access on the previous hits in the *lasthit_arr* array and because each thread has its own *lasthit_arr* when pursuing coarse-grained parallelism for BLASTP, coalesced memory access when the threads of a warp are used for different sequence alignments proves to be effectively impossible.

Even a straightforward *fine-grained* multithreaded approach that uses multiple threads to unfold the "for" loop in Algorithm 1 can also lead to severe branch divergence on a GPU. Why? Due to the uncertainty in both the number of hits on different words and the distance to previous hits along the diagonals. Furthermore, since any position in the *lasthit_arr* array can be accessed during any one iteration, this approach can also cause significant memory-access conflicts. Thus, designing an effective fine-grained parallelization of BLASTP that fully utilizes the capability of the GPU is a daunting challenge. To address this, we decouple the phases of the BLASTP algorithm, use different strategies to optimize each of them, and propose a "binning-sorting-filtering" approach to reorder memory accesses and eliminate branch divergence, as articulated in the following subsections.

3.2 Hit Detection with Binning

We first separate the phases of hit detection and ungapped extension into their own kernels. In our fine-grained hit detection, we use multiple threads to detect consecutive words in a subject sequence and to ensure coalesced memory access. In addition, because ungapped extension executes along the diagonals, we re-organize the output results of the hit-detection phase into diagonal-major order and introduce a binning data structure and bin-based algorithms to bridge the phases of hit detection and ungapped extension. Specifically, we allocate a contiguous buffer in global memory and logically organize this buffer into bins (which will map onto the diagonals) to hold the hits. While one bin could be allocated for one diagonal, we allocate one bin for multiple diagonals to reduce memory usage on the GPU and to allow longer sequences to be handled.

Fig. 5 illustrates our approach to fine-grained hit detection, where *each* word in the subject sequence is scheduled to one thread. A thread retrieves a word from the corresponding position (i.e., column number) in the subject sequence, searches the word in the DFA to get the hit positions (i.e., row numbers), and immediately calculates the diagonal numbers as the difference in corresponding column number and row number. For example, thread 3 retrieves word ABC from column 3 of the subject sequence, searches for ABC in the DFA to get hit positions 1, 7, and 11, and calculates the diagonal numbers as 2, -4, and -8, respectively. Since multiple threads can write hit positions into the same bin simultaneously, we must use atomic operations to address write conflicts, and in turn, ensure correctness.

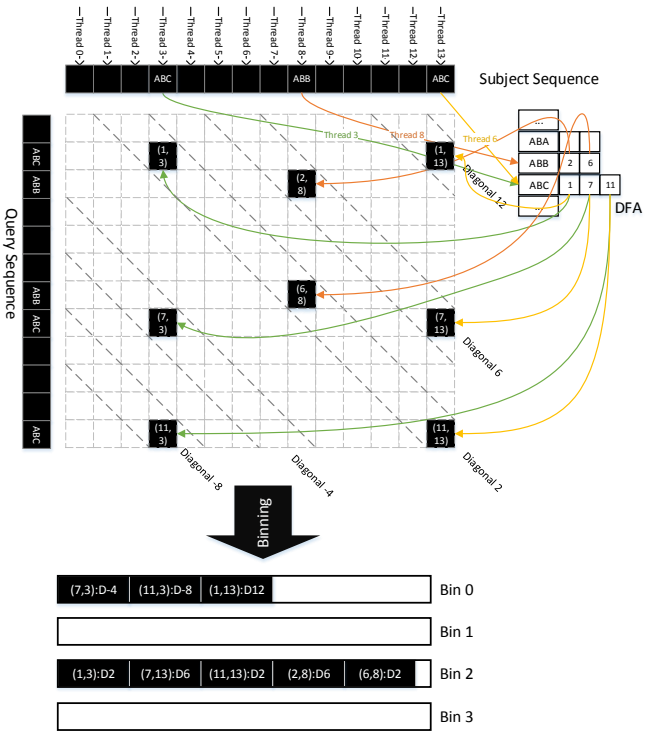


Fig. 5: Hit Detection and Binning

Algorithm 2 describes our fine-grained hit detection algorithm. The variable *num_bins* represents the number of bins, which is a configurable parameter. The algorithm schedules a warp of threads for a sequence based on *warpId*. The word *seq[i][j]* in position j of sequence *seq[i]* is handled by the thread with the *laneId* j. For each hit of the word, the diagonal number is calculated and mapped to a bin on Line 16.

The *top* array stores the currently available position in each bin. Using atomic operations on the *top* array in shared memory, we avoid the heavyweight overhead of atomic operations on global memory. The warp is then scheduled to handle the next sequence after all words in the current sequence are processed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

Algorithm 2 Warp-based Hit Detection

Input: $database$: sequence database;
DFA: DFA lookup table base on query sequence
Output: $bins$: diagonal-based bins that store hits

```
1: tid ← blockDim.x * blockIdx.x + threadIdx.x
2: //calculate total number of warps
3: numWarps ← gridDim.x * blockDim.x/warpSize
4: warpId ← tid/warpSize
5: laneId ← threadIdx.x mod warpSize
6: //initialize i with warpId
7: i ← warpId
8: while database has i-th sequence do
9:     //initialize j with laneId
10:    j ← laneId
11:    while i-th sequence has j-th word do
12:        find hits of j-th word in DFA
13:        for all hit_k in hits do
14:            diagonal ← hit_k.sub_pos − j
                                  + query_length
15:            //calculate bin number
16:            binId ← diagonal mod num_bins
17:            //increment hit counts of the bin
18:            curr ← atomicAdd(top[bin_id], 1)
19:            //store the hit into the bin
20:            bins[binId][curr] ← hit_k
21:        end for
22:        //continue j + warpSize-th word
23:        j ← j + warpSize
24:    end while
25:    //continue i + numWarps-th sequence
26:    i ← i + numWarps
27: end while
28: output bins
```

3.3 Hit Reordering

After hit detection, hits are grouped into bins by diagonal numbers. Because multiple threads can write hits from different diagonals into the same bin simultaneously, hits in each bin could interleave. For example, Fig. 5 shows that hits belonging to diagonal 2 and diagonal 6 interleave. Because ungapped extension can only extend continuous hits whose distance is less than a threshold, we need to further reorder the hits in each bin to enable contiguous memory access during the ungapped-extension phase. To achieve this, we propose a hit-reordering mechanism that includes *assembling, sorting, and filtering*. Fig. 6 provides illustrative examples of these three kernels, respectively.

Hit Assembling: Because it is effectively impossible to get an accurate number of hits for each subject sequence before the completion of the hit-detection phase, we allocate the maximally possible size (i.e., number of words in the query sequence) as the buffer size of each bin. Though this leads to unused memory in the bins, it offers the promise of high performance as we can use a segmented sort [3] to sort the hits per bin. To maximize the throughput of the sort, the data must be contiguously stored, even if they belong to different segments. Thus, prior to sorting, we launch a kernel that assembles the hits from different bins into a large but contiguous array, as shown in Fig. 6(a). Each bin is then processed by a block of threads consecutively for coalesced memory access.

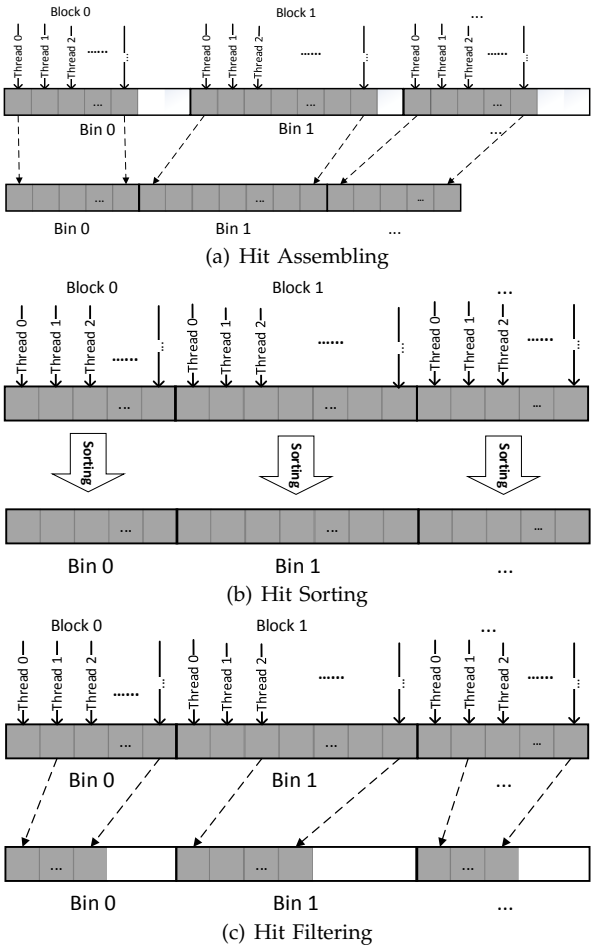


Fig. 6: Three Kernels for Assembling, Sorting, and Filtering

Hit Sorting: A hit includes four attributes: the row number that is the position in the query sequence; the column number that is the position in the subject sequence; the diagonal number that is calculated as the difference of the column number and row number; and the sequence number that is the index of the subject sequence. To unify the attributes and only have to sort once, we propose a bin data structure for the hits. As shown in Fig. 7, we pack the sequence number, diagonal number, and subject position into a 64-bit integer. Because the longest sequence in the most recent NCBI NR database [4] contains 36,805 letters, 16 bits is sufficient to record the subject position and 16 bits for the diagonal number, each of which can represent 64K positions. With this data structure, we sort hits in each bin *once* instead of by the diagonal number and subject position, respectively. This data structure provides an added benefit during ungapped extension — all the required information, including sequence number, query position (i.e., $subject\_position - diagonal\_number$), and subject position can be obtained in one memory access.

Using the segmented sort kernel from the Modern GPU Library [3] by NVIDIA, according to the experiments, we found that as we vary the number of segments for a given data size, the throughput increases as more segments are used. Since the total

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

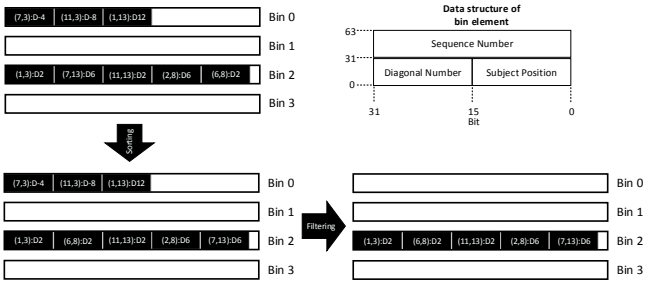


Fig. 7: Sorting and Filtering with Bin Data Structure

number of hits after the hit-detection phase is fixed, we can increase the number of bins to improve sorting performance but at the expense of more memory usage. Because GPU device memory is limited, we must choose an appropriate number of bins that balances sorting performance and memory usage. We set the number of bins as a configurable parameter in our cuBLASTP algorithm because many factors, such as the size of device memory and the query length, affect the choice of the number of bins.

Hit Filtering: With the bins now sorted, we introduce hit filtering to eliminate hits whose distances with neighbors are larger than a specified threshold because these hits cannot trigger ungapped extension. As shown in Fig. 6(c), we use a block of threads to check consecutive hits in each bin for coalesced memory access. A thread scheduled for one hit compares the distance to its neighbor on left. If the distance to the neighbor is less than the threshold, the hit is kept and passed to the ungapped-extension phase.

To avoid global synchronization and atomic operations, we write extendable hits into a dedicated buffer that is maintained by each thread block. The overall performance of this additional filtering step is then determined by the ratio of the overhead of hit filtering over the overhead of branch divergence. (Our experimental results show that only 5% to 11% of the hits from the hit-detection phase are passed to ungapped extension; thus the overall cuBLASTP performance improves due to this hit filtering.)

3.4 Fine-Grained Ungapped Extension

After hit reordering, the hits in each bin are arranged in ascending order by diagonals, and the hits that cannot be used to trigger ungapped extension have been filtered out. Based on the ordered hits, we design a *diagonal-based, ungapped-extension* algorithm, as depicted in Algorithm 3, where *each* diagonal is processed by a thread. So, as shown from Lines 7 to 10, different threads are scheduled to different bins, and threads in a warp are scheduled to different diagonals based on the warp id *warpId*. We then call the *ungapped_ext* function to extend the diagonal until a gap is encountered or the diagonal is ended. The variable *ext* represents the extension result. Because an extension could cover other hits along the diagonal, Line 19 determines if a hit is covered by the previous extension. If the hit is *not* covered by the previous extension, it can be used to trigger an extension. However, the above step introduces divergent branching.

Algorithm 3 Diagonal-based Ungapped Extension

Input: $bins$ binned hits
Output: $extensions$: results of ungapped extension
1: $tid \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2: $numWarps \leftarrow gridDim.x * blockDim.x/warpSize$
3: $warpId \leftarrow tid/warpSize$
4: $laneId \leftarrow threadIdx.x \bmod warpSize$
5: $i \leftarrow warpId$
6: //go through all bins by warps
7: **while** $i < num\_bins$ **do**
8: $j \leftarrow laneId$
9: //process all diagonals in the bin by lanes
10: **while** $j < bin_i.num\_diagonals$ **do**
11: //initialize last extension position
12: $ext\_reach \leftarrow -1$
13: **for all** hit_k in $diagonal_j$ **do**
14: //get hit information
15: $sub\_pos \leftarrow hit_k.sub\_pos$
16: $query\_pos \leftarrow hit_k.sub\_pos - hit_k.diag\_num$
17: $seq\_id \leftarrow hit_k.seq\_id$
18: //check if the pos has been extended
19: **if** $sub\_pos > ext\_reach$ **then**
20: $ext \leftarrow ungapped\_ext(seq\_id, query\_pos, sub\_pos)$
21: $extensions.add(ext)$
22: //update with new extension pos
23: $ext\_reach \leftarrow ext.sub\_pos$
24: **end if**
25: **end for**
26: $j \leftarrow j + warpSize$
27: **end while**
28: $i \leftarrow i + numWarps$
29: **end while**
30: output $extensions$

Due to the above divergent branching, we propose an alternative fine-grained approach to Algorithm 3 called *hit-based ungapped extension*, as shown in Algorithm 4. This approach seeks to improve performance by trading off divergent branching for redundant computation. Specifically, each thread extends a different hit independently. Thus, the results from the ungapped extension of different hits could result in the same output (i.e., redundant computation that results in duplicates). These duplicates are then independently stored on a per-thread basis on Line 14. Unlike Algorithm 3, this algorithm requires a de-duplication step before the remaining phases of gapped extension and alignment with traceback can be run.

Intuitively, which of the two algorithms performs best depends on the characters in the query sequence and the subject sequences. If there are too many hits that will be covered by the extension of other hits in the diagonal, then diagonal-based ungapped extension should perform better; otherwise, hit-based ungapped extension will. However, while hit-based extension eliminates divergent branching, it can create load imbalance. That is, because different hits in one diagonal could be extended to different lengths and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

Algorithm 4 Hit-based Ungapped Extension

Input: bin binned hits
Output: $extensions$: results of ungapped extension

```
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_bins do
7:     j ← laneId
8:     //process all hits in the bin by lanes in parallel
9:     while j < bin_i.num_hits do
10:        sub_pos ← hit_j.sub_pos
11:        query_pos ← hit_j.sub_pos − hit_j.diag_num
12:        seq_id ← hit_j.seq_id
13:        ext ← ungapped_ext(seq_id,
                    query_pos, sub_pos)
14:        extensions.add(ext)
15:        j ← j + warpSize
16:    end while
17:    i ← i + numWarps
18: end while
19: output extensions
```

if (at least) one hit can be extended much longer than other hits, then all other threads in the warp must wait for the completion of the longest extension.

To address the above, we present a window-based extension, as detailed in Algorithm 5. It consists of the following steps: (1) divide a warp of threads into different windows; (2) map one window to one diagonal; and (3) extend hits in a diagonal one by one using a window-sized set of threads at the same time. Because this approach uses a window-sized set of threads to extend a single hit, it can speedup the hit-based extension on the longest extension and reduce the load imbalance that would otherwise more adversely affect performance.

Fig. 8 illustrates how computation proceeds in window-based ungapped extension, along with details on gap detection. A gap can be detected by computing the accumulated score for each extended position from the hit position, and then comparing the score change from the highest score along the extension with a threshold. In this figure, we present two windows, each of which extends the IYP hit along the diagonal but in opposite directions.

For brevity, we only discuss the extension to the IYP hit with the right window; the left window is handled concurrently in a similar fashion. First, we map the window-sized set of threads (in this case, 8) along consecutive positions from the hit and then calculate the prefix sum of each position for the *PrefixSum* array using the optimized scan algorithm derived from the CUB library [1]. This prefix sum in the right window produces the highest score of 12, as circled in the *PrefixSum* array.

Then, each thread after the position with the highest score calculates the score changed from the highest score while the threads before the highest score position simply record the contribution to the highest

Algorithm 5 Window-based Ungapped Extension

Input: bin binned hits, $winSize$ size of windows
Output: $extensions$: results of ungapped extension

```
1: numBlocks ← gridDim.x
2: //get number of windows in a thread block
3: numWin ← blockDim.x/winSize
4: //get window id
5: winId ← threadIdx.x/winSize
6: //get lane id in the window
7: wLaneId ← threadIdx.x mod winSize
8: i ← blockIdx.x
9: //go through all bins by blocks
10: while i < num_bins do
11:     j ← winId
12:     //go through all diagonals in the bin by wins
13:     while j < bin_i.num_diagonals do
14:         ext_reach ← −1
15:         for all hit_k in diagonal_j do
16:             //get hit informatioin
17:             sub_pos ← hit_k.sub_pos
18:             query_pos ← hit_k.sub_pos
                        −hit_k.diag_num
19:             seq_id ← hit_k.seq_id
20:             if sub_pos > ext_reach then
21:                 //perform window-based extension
22:                 ext ← ungapped_ext_win(seq_id,
                     query_pos, sub_pos, wLaneId, winSize)
23:                 //lane 0 stores the extension
24:                 if wLaneId = 0 then
25:                     extensions.add(ext)
26:                 end if
27:                 ext_reach ← ext.sub_pos
28:             end if
29:         end for
30:         j ← j + numWin
31:     end while
32:     i ← i + numBlocks
33: end while
34: output extensions
```

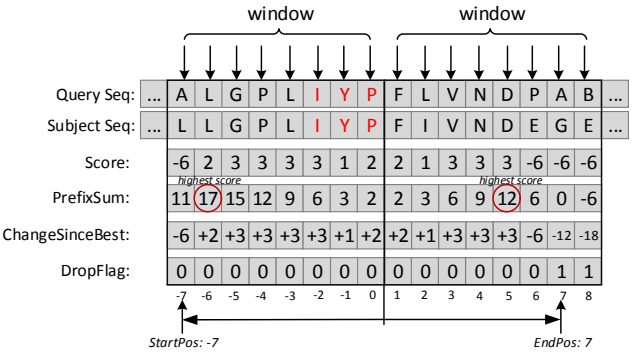


Fig. 8: Example of Window-based Extension. In this example, *dropoff* threshold is -10.

score, i.e., the changes from the previous positions. After this step, our window-based algorithm generates the *ChangeSinceBest*. Next, by comparing to the *dropoff* threshold (i.e.,-10, as noted in the figure), the algorithm then generates the *DropFlag* array. If the change is more than the threshold, a "1" is set to denote this position as a gap; otherwise, a "0" is set. If there is a gap, the algorithm then writes the start position and end position of this extension with the highest score into the output of the ungapped

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

extension. If there is no gap in the window like the left window in the figure, the algorithm goes to the next iteration to move the windows forward. (This figure also illustrates the redundant computation in the window-based ungapped extension: even if the gap exists in the middle of the window, all positions of the window have to be checked.)

Algorithm 5 describes the details of the window-based ungapped extension. Because we use one window on one diagonal to check hits one by one, we still need to check whether the current hit is covered by the previous extension at Line 20. However, this approach removes the redundant computation that would have otherwise been done with our hit-based extension. As a result, we use a configurable parameter to allow the user to select which ungapped extension algorithm to execute at run time: diagonal-based, hit-based, or window-based, as noted in Fig. 9.

3.5 Hierarchical Buffering

To fully utilize memory bandwidth and further improve cuBLASTP performance, we propose a *hierarchical buffering* approach for the core data structure (DFA) used in hit detection. As shown in Fig. 2(a), the 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 words in subject sequences. Loading the DFA into shared memory can improve the data access bandwidth. However, because the number of query positions depends on the length of the query sequence, fetching all positions into the shared memory may affect the occupancy of GPU kernels and offset the improvement from higher data access bandwidth, especially for long sequences. Thus, we load the states that have relatively fixed but small size into shared memory and store the query positions into constant memory.

On the latest NVIDIA Kepler GPU, a 48-kB read-only cache with relaxed memory coalescing rules provides reusable but randomly accessed data. We allocate the query positions in the global memory but tag them with the keyword "const __restrict" for loading them into the read-only cache automatically.

Fig. 10 shows the hierarchical buffering architecture for the DFA on a Kepler GPU. We put the DFA states, e.g., ABB and ABC, into the shared memory. For the first access of ABB from thread 3, the positions are written into bins and loaded into the read-only cache. For the subsequent access of ABB from thread 4, the positions are obtained from the cache.

The PSS matrix is another core data structure that is highly reused in ungapped extension. The number of columns in the PSS matrix is equal to the length of the query sequence, as shown in Fig. 2(b). However, because each column contains 64 bytes (32 rows with 2 bytes for each), the size of the PSS matrix increases

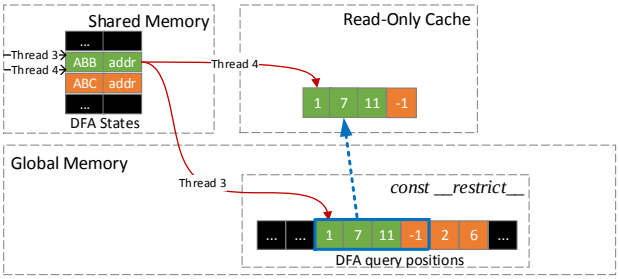


Fig. 10: Hierarchical Buffering for DFA on Kepler GPU

quickly with the query length. The 48-kB shared memory cannot hold the PSS matrix when the query sequence is longer than 768.

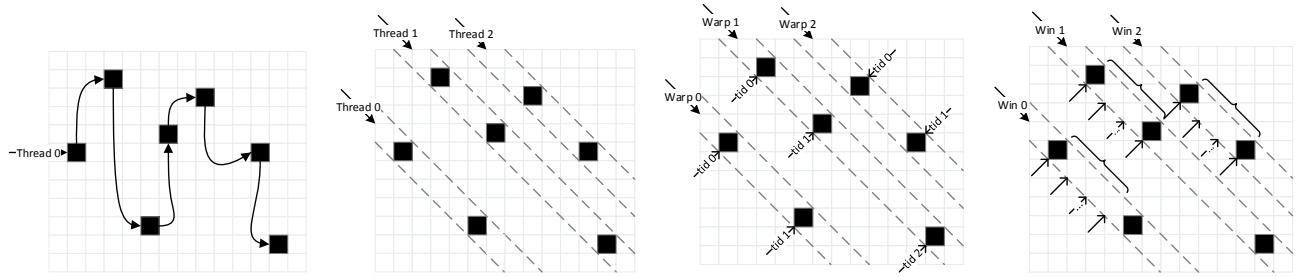
On the other hand, the scoring matrix can be used to substitute the PSS matrix. For example, BLOSUM62 matrix, which consists of 32 * 32 = 1024 elements and has a fixed size of only 2 kB (i.e., 2 bytes per element), can be always put into the shared memory. Therefore, for longer query sequences, the BLOSUM62 matrix in the shared memory can provide better performance, even though more memory operations are needed compared with 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 4.

3.6 Optimizing Gapped Extension and Alignment with Traceback on a Multicore CPU

After the most time-consuming phases of BLASTP accelerated, the remaining phases, i.e., gapped extension and alignment with traceback, now consume the largest percentage of time. Specifically, for a query sequence with 517 characters (i.e., *Query517*), Fig. 11 shows that after applied fine-grained optimizations on GPU, the percentage of time spent on hit detection and ungapped extension is dropped from 80% (FSA-BLAST) down to 52% (cuBLASTP with one CPU). The percentage of time spent on gapped extension and alignment with traceback, however, grows up from 13% to 32% and 5% to 13%, respectively. Thus, it is necessary to optimize these two stages for better overall performance.

In the BLASTP algorithm, only the high-scoring seeds from *un*gapped extension can be passed to the gapped-extension stage. Although the gapped extension on each seed is independent, and the extension itself is compute-intensive, only a small percentage of subject sequences require the gapped extension. If we offload the gapped extension to GPU, CPU will be idle during most of the BLASTP search. In order to improve the resource utilization of the whole system, i.e., make use of both GPU and CPU, parallelize the gapped extension on CPU is an alternative. Furthermore, though there were several studies proposed to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics



(a) Coarse-grained Extension (b) Diagonal-based Extension (c) Hit-based Extension (d) Window-based Extension

Fig. 9: Four Parallelism Strategies of Ungapped Extension

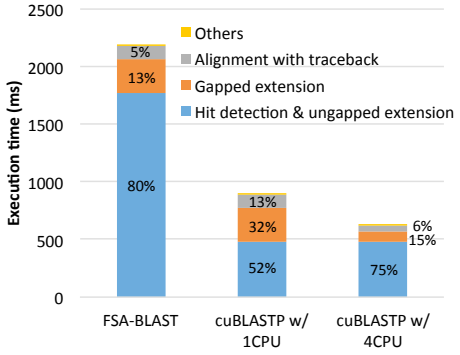


Fig. 11: Time Breakdown for *Query517* on *Swissprot* Database

parallelize the gapped extension on GPU, e.g., CUDA-BLASTP, they had to modify the dynamic programming method of the gapped extension on GPU for the performance. As a result, we optimize the gapped extension on CPU with Pthreads. For the alignment with traceback, due to the data dependency and the random memory access, we also optimize it on CPU with multithreading. In order to reduce the overhead of data transfer between CPU and GPU, we design a pipeline to overlap the computations on CPU and GPU, and the data communication on PCIe. Fig. 12 illustrates the pipeline design. Once the kernels of hit detection and ungapped extension for one block of the database are finished on GPU, the intermediate data is sent back to CPU asynchronously for the remaining phases. At the same time, the kernels for hit detection and ungapped extension are triggered for the next data block. With the pipeline design, we can overlap the computations on CPU and GPU, and the data transfer on PCIe for different data blocks.

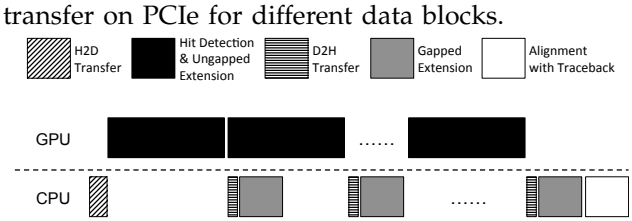


Fig. 12: Overlapping Hit Detection and Ungapped Extension on GPU and Gapped Extension and Alignment with Traceback on CPU

Fig. 11 shows that the multithreaded optimization (cuBLASTP with four CPU threads) significantly improves the gapped extension and the alignment with traceback. Ultimately, the overall performance improvement is more than four-fold over FSA-BLAST. Fig. 13 shows multithreaded gapped extension and alignment with traceback exhibiting strong scaling.

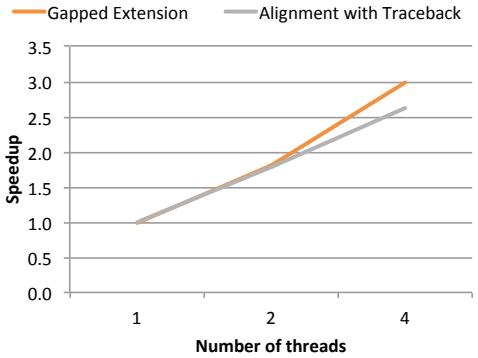


Fig. 13: Strong Scaling for Gapped Extension and Alignment with Traceback on Multicore CPU

4 Performance Evaluation

We conduct our experimental evaluation on a compute node that includes an Intel Core i5-2400 quad-core processor (with 6MB shared L3 cache and 8GB DDR3 main memory) and NVIDIA Kepler K20c GPU. The system runs Debian Linux 3.2.35-2 and NVIDIA CUDA toolkit 5.0. For input data, we use two typical NCBI databases [4]. The first database is *env_nr*, which includes about 6-million sequences whose total size is 1.7 GB and where the average length of the sequences is about 200 letters. The second is *swissprot*, which includes 300,000 sequences with a total size of 150 MB. The average length is 370 letters. For the input query sequences, we choose three sequences, whose lengths are 127 ("query127"), 517 ("query517"), and 1054 ("query1054") bytes, to represent short, medium, and long sequences, respectively.

4.1 Evaluation of Configurable Parameters

We first evaluate the performance of cuBLASTP kernels with different numbers of bins. Fig. 14 shows that the performances of hit sorting and hit filtering can be constantly improved if we increase the number of bins per warp. However, the performance of hit detection drops dramatically after 128 bins. That is, because more bins will use more shared memory to record the current header, and in turn, decrease the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

occupancy of the kernel. Thus, in order to achieve the maximum overall performance, the optimal number of bins per warp should balance the performance of hit detection with hit sorting and filtering. In our experimental environment, we set the number of bins per warp to 128 for the best overall performance.

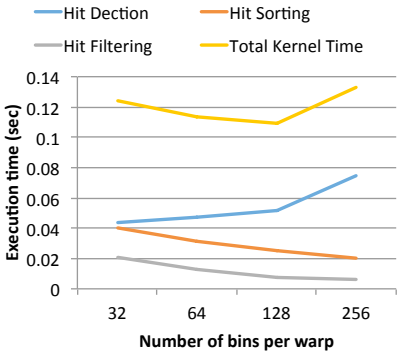


Fig. 14: Execution Time of Different Kernels with Different Numbers of Bins for *Query517* on *Swissprot* Database

Second, in comparing the performance of using the PSS versus BLOSUM62 matrix, Fig. 15 shows that the PSS matrix performs better for the short sequence (query127) whereas the BLOSUM62 matrix performs better for longer sequences (query517 and query 1054), as reasoned and predicted in Section 3.5. In short, we observe a –24%, 50%, and 237% improvement in performance when using the BLOSUM62 matrix. As a result, we configure our algorithm to use the PSS matrix for "query127" and the BLOSUM62 matrix for "query517" and "query1057" on NVIDIA Kepler K20c GPU for the following evaluations.

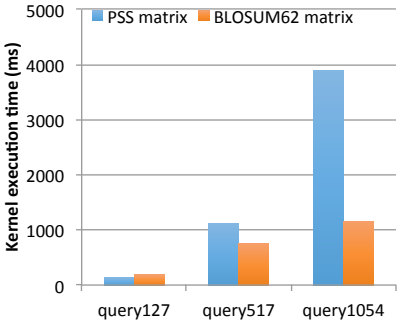
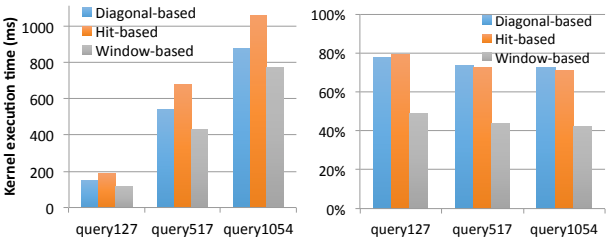


Fig. 15: Performance with Different Scoring Matrices

4.2 Evaluation of our Fine-Grained Algorithms for cuBLASTP: Diagonal-, Hit-, and Window-Based

Fig. 16(a) shows that window-based extension delivers 24%, 20%, and 12% better performance for *query127*, *query517*, and *query1054*, respectively, when compared to diagonal-based extension. Similarly, window-based extension achieves 38%, 36%, and 27% better performance when compared to hit-based extension. Fig. 16(b) compares the divergence overhead of the three algorithms. The window-based algorithm experiences significant improvement in divergence overhead, when compared with the other two algorithms. As a result, we configure our cuBLASTP algorithm to use window-based extension for these two databases on the NVIDIA Kepler K20c GPU in the following evaluations.



(a) Execution Time (b) Divergence Overhead

Fig. 16: Performance Numbers with Different Extensions

Fig. 17 illustrates that cuBLASTP performance can always improve by adopting our hierarchical buffering mechanism, where the read-only cache is used to store the DFA for the hit detection.

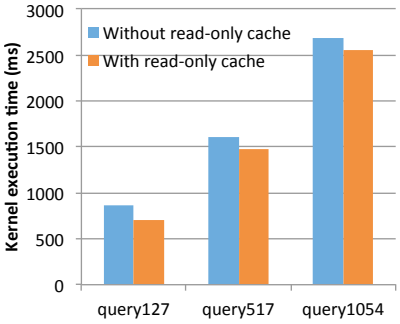


Fig. 17: Performance with and without Read-only Cache

4.3 Performance Comparison to Existing BLASTP Algorithms

Fig. 18 presents the normalized speedup of our fine-grained cuBLASTP over the sequential FSA-BLAST on CPU, the multithreaded NCBI-BLAST on CPU, and the state of the art GPU-based implementations CUDA-BLASTP [29] and GPU-BLASTP [26].

Compared with the single-threaded FSA-BLAST, Fig. 18(a) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to 7.9-fold speedup and 5.5-fold speedup for the critical phases of BLASTP, i.e., hit detection and ungapped extension. Fig. 18(b) shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 3.6-fold and 6-fold, respectively.

Compared with NCBI-BLAST with four threads, Fig. 18(c) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to 2.9-fold speedup and 3.1-fold speedup for the critical phases. Fig. 18(d) shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 2.1-fold and 3.4-fold, respectively.

Compared with CUDA-BLASTP on NVIDIA Kepler K20c GPU, Fig. 18(e) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to a 2.9-fold speedup and 2.1-fold speedup for the critical phases. Fig. 18(f) shows that for the overall performance, including all stages of BLASTP and the data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

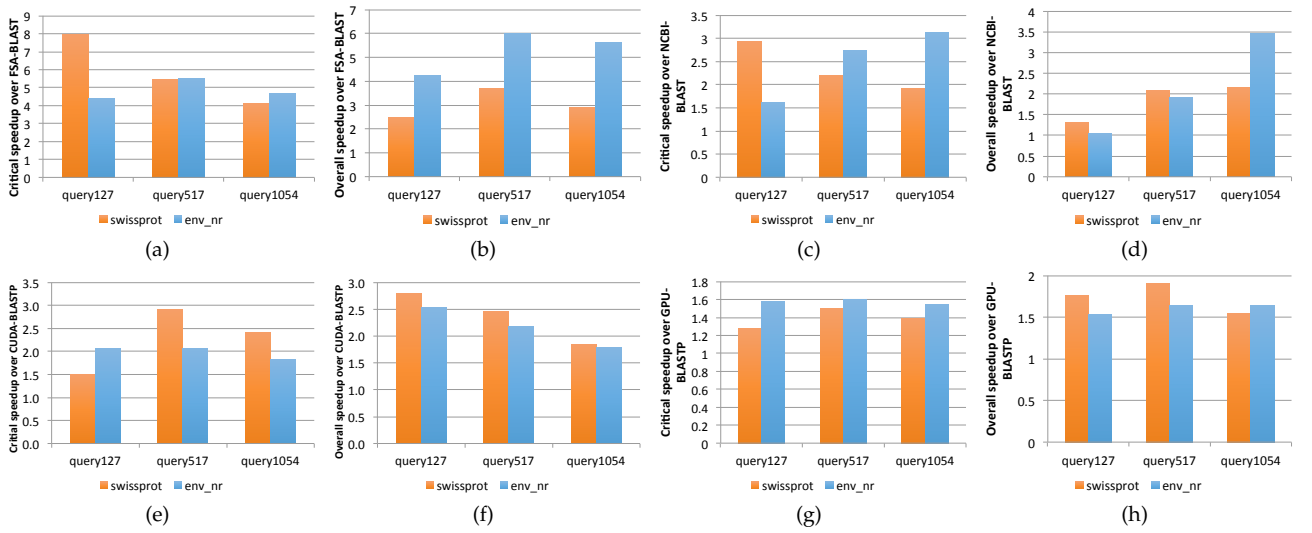


Fig. 18: Speedup for Critical Phases and Overall Performance Respectively of cuBLASTP over FSA-BLAST(a-b), NCBI-BLAST with Four Threads(c-d), CUDA-BLASTP(e-f) and GPU-BLASTP(g-h)

transfer between CPU and GPU, the corresponding performance improvements using cuBLASTP are 2.8-fold and 2.5-fold, respectively.

Finally, with respect to GPU-BLASTP, Fig. 18(g) shows that on the *swissprot* and *env_nr* database, cuBLASTP achieves up to 1.5-fold speedup and 1.6-fold speedup for the critical phases. Fig. 18(h) shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 1.9-fold and 1.6-fold, respectively.

Fig. 19(a), 19(b), and 19(c) show the profiling results of global memory load efficiency, divergence overhead, and occupancy, achieved for cuBLASTP, CUDA-BLASTP, and GPU-BLASTP on NVIDIA Kepler K20c GPU. Because we observed similar results on other query sequences, we only report the results of "query517" for the *env_nr* database.

Fig. 19(a) shows 67.0%, 46.2%, 25.0%, and 81.0% global memory load efficiency for the four respective kernels of cuBLASTP; and only 5.2% for CUDA-BLASTP and 11.5% for GPU-BLASTP, both of them use a single coarse-grained kernel, where both hit detection and ungapped extension are interleaved together. The significantly improved efficiency of our fine-grained kernels comes from the coalesced memory access. In hit detection, threads in the same warp access positions of subject sequences successively. In sorting and filtering, threads in the same warp access hits in each bin successively; and in the window-based ungapped extension, the window-sized set of threads can access successive positions for one hit to calculate the prefix sum and check the score change. In contrast, neither of the coarse-grained kernels of CUDA-BLASTP or GPU-BLASTP can guarantee such coalesced memory accesses.

Fig. 19(b) and 19(c) present the divergence overhead and GPU occupancy, respectively. Our four kernels of cuBLASTP exhibit much lower divergence over-

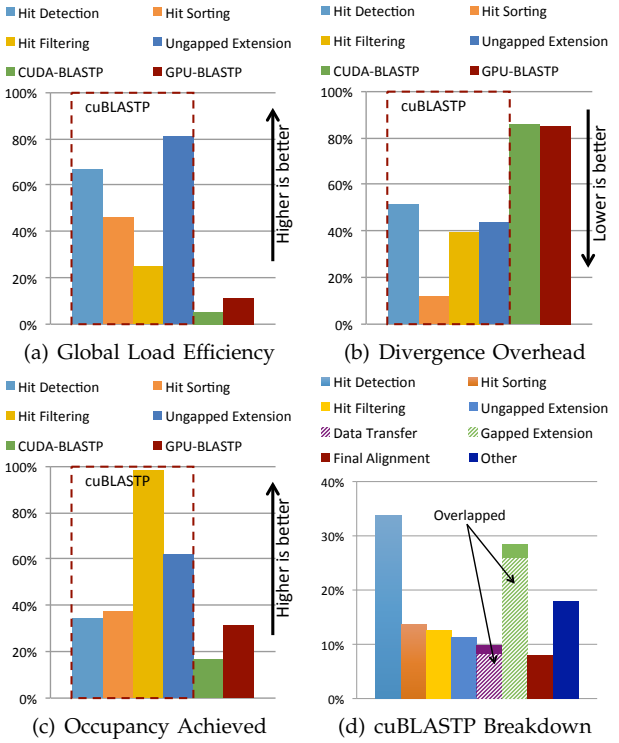


Fig. 19: Profiling on cuBLASTP, CUDA-BLASTP, and GPU-BLASTP

head and higher GPU occupancy than the fused kernels in CUDA-BLASTP and GPU-BLASTP. Fig. 19(d) shows the breakdown of the overall execution time when aligning "query517" on *env_nr* database with cuBLASTP. Although the data transfer between CPU and GPU, and the gapped extension on CPU have the non-negligible execution time, we can overlap them with the kernels running on GPU, as shown in the shadowed bars of this figure. We also find after we optimize all stages of BLASTP on GPU and CPU, the remaining part of BLASTP, denoted as "Other" in this figure, can occupy near 18% total execution time. This part includes the time spent on the database read,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

the DFA and PSS matrix build, and the final results output. We will further investigate the time spent on this part when we extend our research to GPU clusters in the future. Finally, we would like to mention that the output of cuBLASTP is *identical* to the output of FSA-BLAST.

5 Related Work

Many studies have been conducted to parallelize BLAST tools on different parallel architectures because of its compute- and data-intensive nature. NCBI BLAST+ [8] uses pthreads to speedup BLAST on a multicore CPU. On CPU clusters, TurboBLAST [24], ScalaBLAST [10], and mpiBLAST [14] have been proposed. Among them, mpiBLAST is a widely-used one based on NCBI BLAST. With efficient task scheduling and scalable I/O subsystem, mpiBLAST can leverage tens of thousands processors to speedup BLAST.

To achieve higher throughput on a per-node basis, BLAST has also been mapped and optimized onto various accelerators, such as FPGAs [5], [16], [6] and GPUs [21], [9], [23], [29], [26], [17]. Relative to FPGAs, the work of Mahram et al. [6] is notable for its co-processing approach, which leverages both the CPU and FPGA to accelerate BLAST. In general, FPGAs accelerate BLAST search by pre-filtering dissimilar subject sequences rather than speeding up the actual algorithm itself. That is, the FPGA eliminates dissimilar sequences and generates a pre-filtered database, i.e., reduced search space, which the CPU then performs the BLAST search upon.

For GPU accelerators, CUDA-BLASTN [21], CUDA-NCBI-BLAST [9], GPU-BLAST [23], CUDA-BLASTP [29], GPU-BLASTP [26], and G-BLASTN [17] have all been proposed since 2009. CUDA-BLASTN was the first implementation of BLAST on GPU for nucleotide sequence alignments. After that, CUDA-NCBI-BLAST was published for protein sequence alignment. However, without GPU architecture-specific optimizations, CUDA-NCBI-BLAST only achieved 1.7-fold to 2.7-fold speedup on a NVIDIA G80 GPU over a single-core Pentium 4 CPU. Shortly thereafter, GPU-NCBI-BLAST built on NCBI BLAST was proposed. The most time-consuming phases, including hit detection and ungapped extension, were ported to the GPU. With the same accuracy as NCBI BLAST, the authors reported approximately a 4-fold speedup using a NVIDIA Fermi GPU over a single-threaded CPU implementation and a 2-fold speedup over a multi-threaded CPU implementation on a hexa-core processor. CUDA-BLASTP was proposed to use a compressed DFA for hit detection with an additional step to sort the subject sequences by their lengths to improve the load balance. CUDA-BLASTP also ported the gapped extension on GPU. GPU-BLASTP improved the load balance further via a runtime work-queue design, where a thread could grab the next sequence after processing the current subject sequence. GPU-BLASTP also provided a two-level buffering mechanism, which wrote the output of the ungapped extension first to a local buffer for each thread and then to a global buffer. This mechanism avoided global atomics to write the output of different sequences, whose sizes could not be determined in advance. Based on the results from the survey paper [12], CUDA-BLASTP and GPU-BLASTP are top two fastest GPU implementations of BLAST for protein sequence search. As a result, we compared CUDA-BLASTP and GPU-BLASTP with cuBLASTP proposed in this paper. Most recently, G-BLASTN, based on NCBI-BLAST, was released for nucleotide sequence alignment. With optimizations on GPU and parallelism on CPU, including multithreading and vectorization, G-BLASTN achieves up to a 7-fold overall speedup over the multithreaded NCBI-BLAST for nucleotide sequence search on a quad-core CPU. Because BLASTN has already been implemented as a fine-grained algorithm, G-BLASTN did *not* have the challenges of BLASTP when mapped to GPU architectures.

Finally, we note that this paper is most closely related to our previous research [15]. This paper, however, differs from the previous work in the following ways: (1) we further reduce the divergence overhead in the ungapped extension via a new algorithm, i.e., window-based parallelism; (2) we apply a sophisticated segmented sort to optimize hit sorting and filtering; and (3) we optimize the gapped extension and alignment with traceback on CPU with multithreading; and (4) we overlap and pipeline the computations on the GPU and CPU as well as the data transfer between GPU and CPU for better performance.

6 Conclusion

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. To significantly reduce the branch divergence and irregular memory access, we propose binning-sorting-filtering optimizations to reorder memory accesses in the BLASTP algorithm. Our algorithms for diagonal-based and hit-based ungapped extension further reduce branch divergence and improve performance. Finally, we also propose a hierarchical buffering mechanism for the core data structures, which takes advantage of the latest NVIDIA Kepler architecture.

We optimize the remaining phases of cuBLASTP on a multicore CPU with pthreads. On a compute node with a quad-core Intel Sandy Bridge CPU and a NVIDIA Kepler GPU, cuBLASTP achieves up to a 7.9-fold and 3.1-fold speedup over single-threaded FSA-BLAST and multithreaded NCBI-BLAST with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2015.2489662, IEEE/ACM Transactions on Computational Biology and Bioinformatics

four threads for the critical phases of cuBLASTP, namely hit detection and ungapped extension, and up to a 6-fold and 3.4-fold speedup for the overall performance, respectively. Compared with CUDA-BLASTP, cuBLASTP delivers up to a 2.9-fold and 2.8-fold speedup for the critical phases of cuBLASTP and for the overall performance, respectively. Finally, compared with GPU-BLASTP, cuBLASTP delivers up to a 1.6-fold and 1.9-fold speedup for the critical phases of cuBLASTP and for the overall performance, respectively.

In summary, our research with cuBLASTP consists of a novel fine-grained method for optimizing a critical life sciences application that has irregular memory-access patterns and irregular execution paths on a single compute node having CPU and GPU. In the future, we plan to extend our research for very large databases on GPU clusters. Our preliminary research with mpiBLAST [14] revealed that the result sorting, merging, and ranking from multiple nodes could become a time-consuming step, which in turn, would be the performance bottleneck on GPU clusters after the techniques proposed in this paper are applied to address irregular compoutation. In addition, we seek to generalize our optimizations and apply them to other irregular applications on GPU and manycore accelerators, such as Intel Xeon Phi.

ACKNOWLEDGMENT

This research was supported in part by NSF IIS-1247693 (BIGDATA) and NSF CNS-0960081 (MRI), which resulted in the GPU-accelerated HokieSpeed supercomputer, operated by Advanced Research Computing at Virginia Tech.

REFERENCES

[1] CUB Project. http://nvlabs.github.io/cub/.
[2] FSA-BLAST. http://sourceforge.net/projects/fsa-blast/.
[3] Modern GPU. http://nvlabs.github.io/moderngpu/.
[4] NCBI Genbank. ftp://ftp.ncbi.nlm.nih.gov/genbank/.
[5] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Trans. Reconfig. Tech. and Syst.*, 1(2), 2008.
[6] A. Mahram, and M. C. Herbordt. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-based Prefiltering. In *24th ACM Int'l Conf. on Supercomputing*, 2010.
[7] A. Morgulis, G. Coulouris, Y. Raytselis, T. L. Madden, R. Agarwala, and A. A. Schffer. Database Indexing for Production MegaBLAST Searches. *Bioinformatics*, 24(24):2942, 2008.
[8] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. S. Papadopoulos, K. Bealer, and T. L. Madden. BLAST+: Architecture and Applications. *BMC Bioinformatics*, 10:421, 2009.
[9] C. Ling, and K. Benkrid. Design and Implementation of a CUDA-compatible GPU-based Core for Gapped BLAST Algorithm. In *10th Int'l Conf. on Computational Science*, 2010.
[10] 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):740–749, 2006.
[11] D. A. Benson, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. GenBank. *Nucleic Acids Research*, 42:D32–7, 2014.
[12] D. Glasco. An Analysis of BLASTP Implementation on NVIDIA GPUs. Technical report, Stanford University, 2012.
[13] H. E. Williams. Cafe: An Indexed Approach to Searching Genomic Databases. In *21st Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, New York, NY, USA, 1998.
[14] H. Lin, X. Ma, W. Feng, and N. F. Samatova. Coordinating Computation and I/O in Massively Parallel Sequence Search. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):529–543, 2011.
[15] J. Zhang, H. Wang, H. Lin, and W. Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU. In *29th IEEE Int'l Parallel & Distrib. Processing Symp.*, 2014.
[16] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation. In *19th IEEE Int'l Parallel & Distrib. Proc. Symp.*, 2005.
[17] K. Zhao, and X. Chu. G-BLASTN: Accelerating Nucleotide Alignment by Graphics Processors. *Bioinformatics*, 30:1384–1391, 2014.
[18] M. Cameron. *Efficient Homology Search for Genomic Sequence Databases*. PhD thesis, School of Computer Science and Information Technology, RMIT University, Nov 2006.
[19] M. Cameron, H. E. Williams, and A. Cannane. Improved Gapped Alignment in BLAST. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 1(3):116–129, 2004.
[20] M. Cameron, H. E. Williams, and A. Cannane. A Deterministic Finite Automaton for Faster Protein Hit Detection in BLAST. *Journal of Computational Biology*, 13(4):965–978, 2006.
[21] N. Wan, H. Xie, Q. Zhang, K. Zhao, X. Chu, and J. Yu. A Preliminary Exploration on Parallelized BLAST Algorithm Using GPU. *Computer Engineering & Science*, 31(11):98–112, 2009.
[22] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2010. Version 3.2.
[23] P. D. Vouzis, and N. V. Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. *Bioinformatics*, 27(2):182–188, 2011.
[24] R. D. Bjornson, A. H. Sherman, S. B. Weston, N. Willard, and J. Wing. TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub. In *16th IEEE Int'l Parallel & Distrib. Processing Symp.*, 2002.
[25] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *J. Molecular Biology*, 215(3):403–410, 1990.
[26] S. Xiao, H. Lin, and W. Feng. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *25th IEEE Int'l Parallel & Distrib. Processing Symp.*, 2011.
[27] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. *J. Molecular Biology*, 147(1):195–197, 1981.
[28] W. J. Kent. BLAT–the BLAST-like Alignment Tool. *Genome Research*, 12(4):656–664, apr 2002.
[29] W. Liu, B. Schmidt, and W. Muller-Wittig. CUDA-BLASTP: Accelerating BLASTP on CUDA-enabled Graphics Hardware. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 8(6):1678–1684, 2011.
[30] Z. Ning and A. J. Cox and J. C. Mullikin. SSAHA: a Fast Search Method for Large DNA Databases. *Genome Res*, 11(10):1725–9, Oct 2001.

Jing Zhang is a Ph.D. candidate in the Dept. of Computer Science at Virginia Tech. He received his B.S. in the College of Computer at the National University of Defense Technology (NUDT), China.

Hao Wang is a Research Associate in the Dept. of Computer Science at Virginia Tech. He received his Ph.D. in the Institute of Computing Technology at Chinese Academy of Sciences and completed his postdoctoral training at the Ohio State University.

Wu-chun Feng is a professor and Elizabeth & James E. Turner Fellow in the Dept. of Comp. Science, Dept. of Electrical and Computer Engineering, Health Sciences, and Virginia Bioinformatics Institute at Virginia Tech.