BIOINFORMATICS ORIGINAL PAPER Vol. 30 no. 10 2014, pages 1384–1391
doi:10.1093/bioinformatics/btu047

Sequence analysis

Advance Access publication January 24, 2014

G-BLASTN: accelerating nucleotide alignment by graphics processors

Kaiyong Zhao[1] and Xiaowen Chu[1,2,*]

[1]Department of Computer Science, Hong Kong Baptist University, Hong Kong, China and [2]Institute of Computational and Theoretical Studies, Hong Kong Baptist University, Hong Kong, China

Associate Editor: Dr. John Hancock

ABSTRACT

Motivation: Since 1990, the basic local alignment search tool (BLAST) has become one of the most popular and fundamental bioinformatics tools for sequence similarity searching, receiving extensive attention from the research community. The two pioneering papers on BLAST have received over 96 000 citations. Given the huge population of BLAST users and the increasing size of sequence databases, an urgent topic of study is how to improve the speed. Recently, graphics processing units (GPUs) have been widely used as low-cost, high-performance computing platforms. The existing GPU-BLAST is a promising software tool that uses a GPU to accelerate protein sequence alignment. Unfortunately, there is still no GPU-accelerated software tool for BLAST-based nucleotide sequence alignment.

Results: We developed G-BLASTN, a GPU-accelerated nucleotide alignment tool based on the widely used NCBI-BLAST. G-BLASTN can produce exactly the same results as NCBI-BLAST, and it has very similar user commands. Compared with the sequential NCBI-BLAST, G-BLASTN can achieve an overall speedup of 14.80X under 'megablast' mode. More impressively, it achieves an overall speedup of 7.15X over the multithreaded NCBI-BLAST running on 4 CPU cores. When running under 'blastn' mode, the overall speedups are 4.32X (against 1-core) and 1.56X (against 4-core). G-BLASTN also supports a pipeline mode that further improves the overall performance by up to 44% when handling a batch of queries as a whole. Currently G-BLASTN is best optimized for databases with long sequences. We plan to optimize its performance on short database sequences in our future work.

Availability: http://www.comp.hkbu.edu.hk/~chxw/software/G-BLASTN.html

Contact: chxw@comp.hkbu.edu.hk

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 27, 2013; revised on January 10, 2014; accepted on January 21, 2014

1 INTRODUCTION

The basic local alignment search tool (BLAST) is one of the most fundamental software tools in bioinformatics for matching biological sequences (Altschul *et al.*, 1990, 1997). Due to the explosive growth of sequence data, improving the speed of BLAST has become increasingly critical. In the last decade, many attempts have been made to design and develop new BLAST software tools for specific hardware (Fei *et al.*, 2008; Jacob *et al.*, 2007; Sotiriades and Dollas, 2007; Zhang *et al.*, 2000) or even parallel supercomputers (Lin *et al.*, 2008). Unfortunately, most researchers do not have access to these hardware platforms. Following the popularity of multicore processors, several BLAST software tools using multiple CPU cores for increased speed have been developed. One good example is the widely used National Center for Biotechnology Information (NCBI) BLAST, which supports multithreading in the preliminary stage of the BLAST algorithm (Camacho *et al.*, 2009). Our experiments on a server with two quad-core Intel Xeon CPUs show that the multithreaded NCBI-BLAST can achieve an average speedup of 3~4X over the sequential version. NCBI-BLAST also supports an indexed Mega-BLAST module, which uses the database index to achieve an approximate speedup of 2~4X (Morgulis, 2008). PLAST is a parallel implementation of BLAST (Nguyen and Lavenier, 2009) that applies a new indexing technique together with SSE instructions and multithreading to achieve better alignment speed. KLAST is an optimized and extended version of PLAST, and includes a module KLASTn to compare two sets of DNA sequences. As compared with NCBI BLASTN, KLASTn can achieve good speedup with comparable sensitivity.

In recent years, Graphics Processing Units (GPUs) have been widely accepted as low-cost, high-performance computing platforms (Owens *et al.*, 2008). Compared with traditional multi-core CPUs, GPUs have much higher computational horsepower and memory bandwidth. Many bioinformatics tools have been accelerated by GPUs in recent years (Dematte and Prandi, 2010; Liu *et al.*, 2012a, b; Lu *et al.*, 2012, 2013; Manavski and Valle, 2008). The significant difference between GPU and CPU architectures has created many challenges in developing highly efficient GPU software (Nickolls, 2007). Without the development of carefully designed parallel algorithms and sophisticated optimizations, the huge potential of GPUs may not be fully realized.

Some GPU-based software tools have been developed for protein sequence alignment. Ling's GPU-based BLAST software can achieve a speedup of 1.7~2.7X, compared with NCBI-BLAST (Ling and Benkrid, 2010). Recently, Vouzis and Sahinidis (2011) developed GPU-BLAST, which can typically achieve acceleration speedup of 3~4X relative to the sequential NCBI-BLAST. The major advantage of GPU-BLAST is that it can produce the same results as NCBI-BLAST.

To the best of our knowledge, we are the first to provide an open-source GPU solution, namely G-BLASTN, for nucleotide sequence alignment that can produce the same results as NCBI-

*To whom correspondence should be addressed.

BLAST. G-BLASTN is developed on top of the NCBI-BLAST source code. It currently supports the 'megablast' and 'blastn' modes of NCBI-BLAST. For brevity, we use BLASTN to refer to the nucleotide blast module of NCBI-BLAST. The major idea behind G-BLASTN is to store a small hash table in the fast GPU cache memory and then scan the DNA database in parallel using all of the available GPU cores. We have overcome several challenges to fully use the GPU horsepower. To achieve significant speedup, some other parts of BLASTN have also been optimized. We evaluate G-BLASTN's performance by running a set of experiments on human and mouse genome databases, as well as a partial of the NCBI nucleotide (nt) collection database. Using a contemporary NVIDIA GTX780 GPU with a cost of $650, G-BLASTN under 'megablast' mode can achieve significant speedups over the multithreaded BLASTN running on 4-core or 8-core CPUs. When running under the more sensitive 'blastn' mode, G-BLASTN also achieves reasonable speedups. When processing a batch of queries, G-BLASTN supports a pipeline mode that can further improve the performance by up to 44%.

The remainder of the article is organized as follows. In Section 2, we briefly review the main algorithms of BLASTN and present our design for G-BLASTN. In Section 3, we present the detailed implementation of G-BLASTN. In Section 4, we present the experimental results. We conclude the article in Section 5.

2 METHODS

2.1 BLASTN algorithms

BLASTN is designed to efficiently search nucleotide databases using a nucleotide query sequence (Camacho *et al*., 2009). BLASTN's high level pseudocode is given in Figure 1, which consists of four stages. The 'setup' stage prepares search options, reads and prepares the query sequence and database sequence and builds the lookup table. The 'scanning' stage performs a preliminary search comprising three steps: seeding, ungapped extensions and gapped extensions. The seeding step scans the database for hits (i.e. a match with some word in the lookup table). The hits are then extended by ungapped alignment. The alignments that exceed a threshold score will go through the gapped extensions. Only the gapped alignments that exceed another threshold score will be saved as 'preliminary' matches. The 'trace-back' stage takes the preliminary matches as input, considering ambiguous nucleotides and finds the locations of insertions and deletions. The 'output' stage displays the alignment results to the user.

BLASTN's efficiency relies on the assumption that any alignment of interest between the query and the database will contain at least one W-gram (i.e. a subsequence of length W), where W is a parameter known as 'BLASTN word size'. In practice, for any given query sequence BLASTN will construct a lookup table that stores the offsets into the query where each possible w-gram occurs, where w is a parameter known as the 'lookup table word size' which is less than or equal to W. Because each letter can be one of $\{A, C, G, T\}$, the lookup table has 4^w entries. The seeding procedure walks through the database sequence to find hits. In each round, it fetches a w-gram, calculates its hash value, looks into the lookup table and records all matched offset pairs (i.e. the pair of offsets of the matched w-gram in the query and database, respectively) if there are any. When W is larger than w, it is not necessary to scan the database letter by letter; instead, BLASTN scans the database in strides. The maximum stride size without missing any match is W-w+1. For extremely long nucleotide databases, the seeding procedure is usually the most time-consuming step.

After all w-gram hits have been found, we must determine whether each hit belongs to a W-gram match. This is done through the mini-extension procedure (*a.k.a.* exact match extension), which extends each w-gram in both the left and right directions to check the existence of exact W-gram matches. The mini-extension step can be time consuming if millions of w-gram hits must be extended. Once we find all of the W-gram hits, the ungapped extension step begins, allowing for mismatches. Ungapped alignments that exceed a threshold score are stored for gapped extension. In the scanning stage, gapped extension returns only the score and extent of the alignment while the number and position of insertions, deletions and matching letters are not stored. During the whole-scanning stage, BLASTN processes the sequence in NCBI-NA2 format, in which each nucleic acid is represented by two bits. Hence, ambiguities cannot be handled. In the trace-back stage, ambiguous nucleotides are restored by converting NCBI-NA2 format into NCBI-NA8, and more sensitive heuristic parameters are used for the final gapped alignment. Finally, the output step formats the results according to the user options and prints the results for the user.

2.2 Design of G-BLASTN

GPUs have become mature, many-core processors with much higher computational power and memory bandwidth than today's CPUs. A GPU consists of a scalable number of streaming multiprocessors (SMs), each containing some streaming processors (SPs), special function units (SFUs), a multithreaded instruction fetch and issue unit, registers and a read/write shared memory. CUDA is currently the most popular programming model for general purpose GPU computing. The best way to use the hundreds to thousands of GPU cores is to generate a large number of CUDA threads that can access data from multiple memory spaces during their execution, as illustrated in Figure 2. Each thread has its private registers and local memory. Each GPU kernel function generates a grid of threads that are organized into thread blocks. Each thread block has shared memory visible to all threads within the block and with

```
1     [Setup] prepare the BLASTN options, query, database, lookup table
2     [Scanning] for each of N threads {
2.1     while the database still has unsearched sequences {
2.2       Retrieve a group of sequences from the database
2.3       Seeding: find exact word matches
2.4       Ungapped extensions
2.5       Gapped extensions
2.6     }
2.7   }
3     [Trace-back] for each database sequence containing alignments,
      perform trace-back
4     [Output] print the alignment results
```

Fig. 1. High level Pseudocode of BLASTN

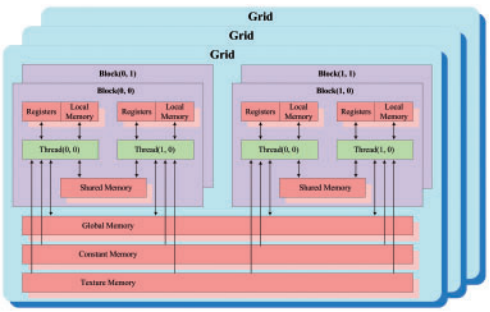

Fig. 2. GPU memory hierarchy

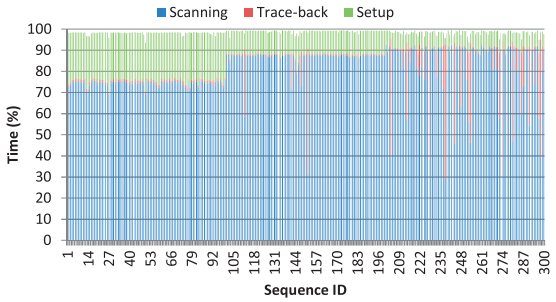

Fig. 3. Profiling of BLASTN for 300 query sequences (500~100 000 bases) against human build 36 genome under 'megablast' mode. The lengths of the query sequences can be found in Supplementary Figure S1

the same lifetime as the block. All threads have access to the same global memory. Two additional read-only memory spaces are accessible by all threads: the constant and texture memory spaces, both of which have limited caches.

Due to the complexity of BLASTN software, exploiting GPUs to accelerate BLASTN is a non-trivial task. The main challenge is that not all of the steps involved in BLASTN are suitable to be parallelized by GPUs. To identify which steps should be parallelized, we conducted a profiling study by running 300 different queries with a broad range of lengths against the human build 36 genome database to analyze the time distribution of different BLASTN steps under 'megablast' mode (Fig. 3). We mainly observed the following details. The scanning stage is the most time-consuming and accounts for 69–93% of the total execution time. Surprisingly, BLASTN spends 5–25% of the total execution time in the setup stage, mainly initializing the mask database. The trace-back stage takes negligible time for most queries, but can occasionally take a very long time.

To achieve a good overall speedup, we designed G-BLASTN as follows. Its major component is a set of CUDA kernel functions that run on GPUs to significantly accelerate the seeding and mini-extension steps in the scanning stage. It is designed to initialize the mask database once and then serve a large number of queries. Therefore, the time spent in database initialization can be largely removed. We optimized the two most time consuming functions in the trace-back stage and further designed a pipeline mode under which the trace-back, output and scanning stages can run simultaneously. The general framework of G-BLASTN is shown in Figure 4.

3 IMPLEMENTATION

We use CUDA C language to implement G-BLASTN based on NCBI BLAST 2.2.28 software package. In the following, we present the detailed implementation of the major modules of G-BLASTN.

3.1 Accelerating the seeding step by GPU

The main task of the seeding step is to scan the database sequences and identify all *w*-gram matches. Due to the large database sizes, the seeding step is the most time consuming in BLASTN. Fortunately, the seeding step can be parallelized due to the independence of the tasks at different offsets of the database. G-BLASTN first loads the database sequences to GPU global memory. Then for each query sequence, it stores a copy of the lookup table in GPU texture memory to achieve ultrafast table lookup. To scan each single database sequence, many GPU

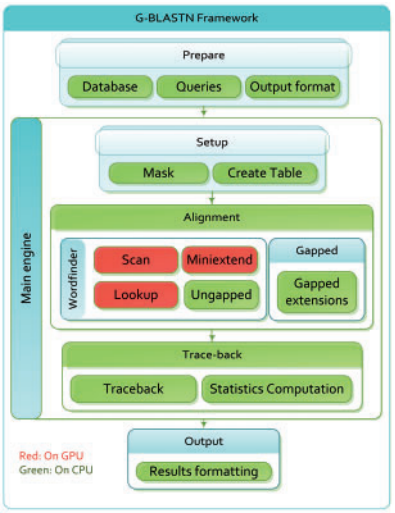

Fig. 4. The framework of G-BLASTN

threads are generated to fully exploit the large number of GPU cores. The number of threads is equal to the number of thread blocks multiplied by the number of threads per thread block. As a rule of thumb, the number of thread blocks should be a multiple of the number of physical SMs available in the GPU. We empirically fine-tune the thread dimensions to optimize the performance.

The implementation of the seeding step on the GPU is a major challenge, however. In CUDA, each thread block is organized as a number of warps, and each warp of threads is executed by a Single Instruction, Multiple Data (SIMD) unit. When threads within a warp take different execution paths, the SIMD unit will take multiple runs to go through these divergent paths, which will significantly decrease the utilization of GPU cores. In the case of BLASTN, the *w*-grams at different offsets of the database sequence may have no match or many matches to the query sequence, which can lead to severe thread branch divergence that decreases the performance significantly. To conquer this challenge, we divide the seeding step into two sub-steps: 'scan' and 'lookup'. In the scan sub-step, we go through the whole-database sequence in parallel and record all offsets of the database that have at least one match to the query. Notice that we do not need to know how many matches have been found and where they are for each offset. Thus, each GPU thread can perform almost the same execution path and the effect of thread branch divergence can be minimized. In the lookup sub-step, we use another GPU-kernel function to recheck all matched offsets and construct the complete set of matched offset pairs. This strategy works very well because the scan sub-step dominates the time of the seeding step.

There is yet another challenge in implementing the scan sub-step on the GPU. Once a thread finds a *w*-gram match, it increases a global counter and writes the matched offset into a global array, resulting in two negative consequences: (i) increasing the global counter is an atomic operation, which means only one among all threads can operate while others have to wait; and (ii) writing a single offset pair into the global array can waste a lot of GPU memory bandwidth. To overcome this challenge, we

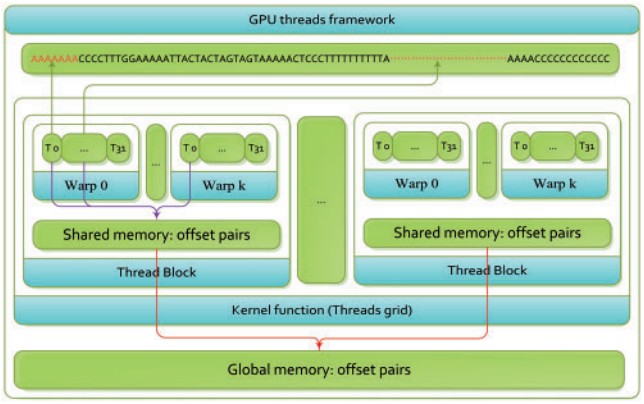

Fig. 5. Framework of scan sub-step on GPU

use a local counter and a local array for each thread block as temporary storage for the global counter and global array. The local counter and array are held in GPU shared memory. Now all thread blocks can operate on the local counters and arrays simultaneously, boosting the overall performance. Once a local array becomes full, the set of offset pairs is written into global array as a whole and the global counter is updated by an atomic operation. We exploit coalesced memory write operations to improve memory throughput. Meanwhile, the number of atomic operations on the global counter can be significantly reduced. The framework of the scan sub-step on GPU is shown in Figure 5.

For performance consideration, BLASTN supports two types of lookup tables for different types of queries: small and megablast (in current NCBI-BLAST, both types of lookup tables are supported by 'blastn' mode and 'megablast' mode). Each type of lookup table has its own set of algorithms. Therefore, we have to implement different GPU-kernel functions for different types of lookup tables.

A small lookup table contains a simple backbone array and an overflow array, both of which are simply an array of 16-bit integers. If the value of a backbone cell is nonnegative, it means that position in the lookup table contains exactly one query offset, which equals the cell value. If the value is -1, the corresponding w-gram does not exist in the query sequence. If the value is $-x$ ($x > 1$), the corresponding w-gram appears multiple times in the query sequence and their offsets begin at offset x of the overflow array and continue until a negative value is encountered. The pseudocode of our GPU scan and lookup kernel functions using a small lookup table are shown in Figures 6 and 7, respectively. The backbone array is held in GPU texture memory. Notice that a GPU-kernel function specifies the behavior of a single GPU thread. There are hundreds of thousands of GPU threads simultaneously active, each of which executes the same instructions while working on different data items.

The megablast lookup table comprises three arrays: presence vector (PV array), hash table (hashtable[]) and next position (next_pos[]). The PV array is a bit field with one bit for each hash table entry. If a hash table entry contains a query offset, the corresponding bit in the PV array is set. The scanning process first checks the PV array to see whether there are any query

Input: backbone[] // *in texture memory*
Output: P1[], P2[], globalCounter // *P1 stores exact offset pairs, P2 stores overflow offset pairs, globalCounter stores the number of matches*
Key Variables: BlastOffsetPair localArray[K]; // *in shared memory*
uint localCounter; // *in shared memory*

```
1   s_index = blockIdx.x*blockDim.x + threadIdx.x;
2   do
3     load base pairs into s from database sequence;
4     h = hash_function(s);
5     hv = backbone[h];
6     calculate db_offset;
7     if hv > −1 then
8       atomicAdd(localCounter, 1);
9       write offset pair (hv, db_offset) into localArray;
10    end if
11    if hv < −1 then
12      atomicAdd(overflowCounter, 1);
13      write offset pair (-hv, db_offset) into P2;
14    end if
15    __syncthreads( ); // local barrier
16    if localCounter >= K/2 then
17      if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
18      __syncthreads( ); // local barrier
19      copy the offset pairs in localArray to P1;
20      if threadIdx.x == 0 localCounter = 0;
21      __syncthreads( ); // local barrier
22    end if
23    update s_index;
24  repeat until out of range
25  if localCounter > 0 then
26    if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
27    __syncthreads( ); // local barrier
28    copy offset pairs in localArray to P1;
29  end if
```

Fig. 6. The GPU scan-kernel function using small lookup table

Input: P1[], P2[], overflowTable[], globalCounter // *P1 is exact offset pair array, P2 is overflow offset pair array, overflowTable is in texture memory*
Output: P1[], globalCounter;

```
1   index = blockIdx.x*blockDim.x + threadIdx.x;
2   read pair (hv, db_offset) from P2[index];
3   q_offset = overflowTable[hv++]; // overflow table lookup
4   do
5     atomicAdd(globalCounter, 1);
6     write offset pair (q_offset, db_offset) into P1;
7     if hv <= the length of overflow table then
8       q_offset = overflowTable[hv++];
9     else
10      break;
11    end if
12  repeat until q_offset < 0
```

Fig. 7. The GPU lookup kernel function using small lookup table

offsets in a particular lookup table entry. The hashtable[] array is a thick backbone with one word for each of the lookup table entries. If a lookup table entry has no query offsets, the corresponding entry in hashtable[] is zero; otherwise, it is an offset into

```
Input: PV // presence vector, in texture memory
Output: P[], globalCounter // P stores all matched offset pairs
Key Variables: BlastOffsetPair localArray[K]; // in shared memory
               uint localCounter; // in shared memory
1   s_index = blockIdx.x*blockDim.x + threadIdx.x;
2   do
3     load base pais into s from database sequence;
4     h = hash_function(s);
5     if BlastMBLookupHasHits(h) == 1 then
6       calculate db_offset;
7       atomicAdd(localCounter, 1);
8       write offset pair (h, db_offset) into localArray
9     end if
10    __syncthreads( ); // local barrier
11    if localCounter >= K/2 then
12      if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
13      __syncthreads( ); // local barrier
14      copy offset pairs in localArray to P;
15      if threadIdx.x == 0 localCounter = 0;
16      __syncthreads( ); // local barrier
17    end if
18    update s_index;
19  repeat until out of range
20  if localCounter > 0 then
21    if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
22    __syncthreads( ); // local barrier
23    copy offset pairs in localArray to P;
24  end if
```

Fig. 8. The GPU scan-kernel function using megablast lookup table

```
Input: P, hashtable, next_pos
Output: P1
1   index = blockIdx.x*blockDim.x + threadIdx.x;
2   read pair (h, db_offset) from P[index];
3   q_offset = hashtable[h];
4   while q_offset > 0
5     atomicAdd(globalCounter, 1);
6     write (q_offset-1, db_offset) to P1;
7     if q_offset < the length of next_pos table then
8       q_offset = next_pos[q_offset];
9     else
10      break;
11    end if
12  end while
```

Fig. 9. The GPU lookup kernel function using megablast lookup table

next_pos[]. The position in next_pos[] is in fact the query offset, and the actual value at that position is a pointer to the succeeding query offset in the chain. A value of zero means the end of the chain. The pseudocode of our GPU scan and lookup kernel functions using the megablast lookup table are shown in Figures 8 and 9, respectively. The scan-kernel function checks the PV array to quickly determine whether there is a match. To achieve the best table lookup performance, the PV array is held in texture memory. The lookup kernel function takes the output of scan function as input and checks the hashtable[] and next_pos[] to find the complete set of matched offset pairs.

```
1   set pointer p_buf to the address of 128-bit data;
2   __m128i t_buf = _mm_loadu_si128(p_buf); // load data into register
3   t_buf=_mm_shuffle_epi8(ntob_table, t_buf); // translate the data
4   _mm_storeu_si128(p_buf,  t_buf); // write back data
```

Fig. 10. SSE instructions used by *s_SeqDBMapNcbiNA8ToBlastNA8*()

3.2 Accelerating the mini-extension step by GPU

It is not uncommon for the scan sub-step to return millions of seed matches. The mini-extension step is designed to verify whether each w-gram match can be extended to a W-gram match when $w < W$. We can create a huge number of GPU threads to extend those w-gram matches simultaneously. Each GPU thread reads one offset pair from the matched offset pair array, extends on the left side and then extends on the right side. If it finds a W-gram match, this offset pair will be recorded for further gapped extension. Given that the mini-extension algorithm exhibits no big difference from the original BLASTN, we do not provide the pseudocode here. We note that there are two versions of mini-extension, one for the small lookup table and another for the megablast lookup table.

3.3 Optimizing the trace-back step

As mentioned in Section 2.2, occasionally the trace-back step takes quite a long time, which may counteract the speedup achieved by the previous steps. Unfortunately, the trace-back step is not naturally suitable for GPUs. We therefore resort to the following optimization techniques. First, function *s_SeqDBMap NA2ToNA8*() uses a translation table to convert sequence data from NCBI-NA2 to NCBI-NA8 format. BLASTN translates the data character by character, which does not fully use the CPU memory bandwidth. In G-BLASTN, we replace four 8-bit memory writes with a single 32-bit memory write, which boosts the speed by two to three times. Second, function *s_SeqDBMapNcbiNA8To BlastNA8*() uses a 16-byte translation table to convert sequence data from NCBI-NA8 to BLAST-NA8 format, character by character. G-BLASTN uses a 128-bit union (denoted by *ntob_table*) to hold the 16-byte translation table, and then SSE instructions to write 16 bytes as a whole, which achieves a speedup of 3∼4X. The SSE instructions are shown in Figure 10.

3.4 Pipeline mode for multiple queries

Once we have accelerated the scanning stage by GPU, other stages such as trace-back and output may start to occupy a relatively large portion of the total execution time, especially when there are many final hits. G-BLASTN supports a pipeline mode when handling a batch of queries, as shown in Figure 11. The main advantage is that GPU and CPU can work on different tasks simultaneously. When the GPU is busy seeding, the CPU can execute the trace-back or output steps for a previous query. To achieve this purpose, G-BLASTN uses multithreading to maintain four queues: query, job, prelim and result. A master thread reads the queries and puts them into the query queue, and

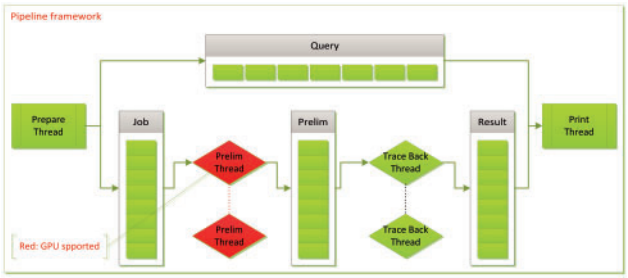

Fig. 11. The pipeline mode of G-BLASTN

Table 1. System configuration

CPU	Memory	GPU	Storage	OS
Intel Core i7-3820 (4-core, 3.6 GHz)	32GB (DDR3 1600)	Nvidia GTX780	SATA 2TB	CentOS 6.4 (Linux kernel 2.6.32)
2 x Intel Xeon E5620 (8-core, 2.4 GHz)	24GB (DDR3 1333)	N/A	SATA 1TB	Redhat 5.5 (Linux kernel 2.6.18)

then creates the job queue. The prelim thread(s) fetches jobs from the job queue and uses GPU to execute the preliminary search, storing the results in the prelim queue. The trace-back thread(s) reads from the prelim queue, executes the trace-back step and stores the results in the results queue. Finally, the print thread prints the results.

4 RESULTS

4.1 General setup and data*sets*

The GPU experiments were performed on a desktop computer with an Intel quad-core CPU and Nvidia GTX780 GPU. The CPU experiments were performed on two different platforms: a 4-core platform which is the same computer that runs the GPU experiments; and an 8-core platform which is a server with two Intel Xeon CPUs. The detailed system configuration is shown in Table 1.

We used the following two command lines to run NCBI BLASTN and G-BLASTN, respectively.

$**blastn** -db <database> -query <query> -task megablast|blastn -outfmt 7 -out <file> -dust yes -window_masker_db <masker_db> -num_threads <1|4|8>

$**gblastn** -db <database> -query_list <query list> -task megablast|blastn -outfmt 7 -out <file> -dust yes -window_masker_db <masker_db> -use_gpu true -mode <1|2> -num_threads <1|4|8>

We used *gettimeofday*() functions to measure the program execution time. Each experiment was run 10 times and the average results are reported in this article.

Databases: We chose human build 36 and mouse build 36 genome databases for the experiments of 'megablast' mode. In addition, we constructed a database to test the 'blastn' mode by selecting all sequences with length no <2 million from NCBI nt database. This partial NCBI nt database has a raw size of 8.4GB and can well fit into a single GPU card with 3GB memory after compression. All databases were masked with WindowMasker (Morgulis *et al.*, 2006a), including low-complexity filtering by DUST (Morgulis *et al.*, 2006b).

Queries: To test the 'megablast' mode, we chose queries from the NCBI ftp server: ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/queries (Morgulis, 2008). Six query sets, each containing 100 queries, were used, which are referred to as Qsmall (~500 bases, range: 501–506), Qmedium (~10 KB, range: 10 000–10 446) and Qlarge (~100 KB, range: 100 001–102 087). To test the 'blastn' mode, we chose the first 500 bacterial sequences from the NCBI server: http://www.ncbi.nlm.nih.gov/sra/SRX338063, namely Qbac.

The length information of all database and query sequences can be found in the Supplementary Material.

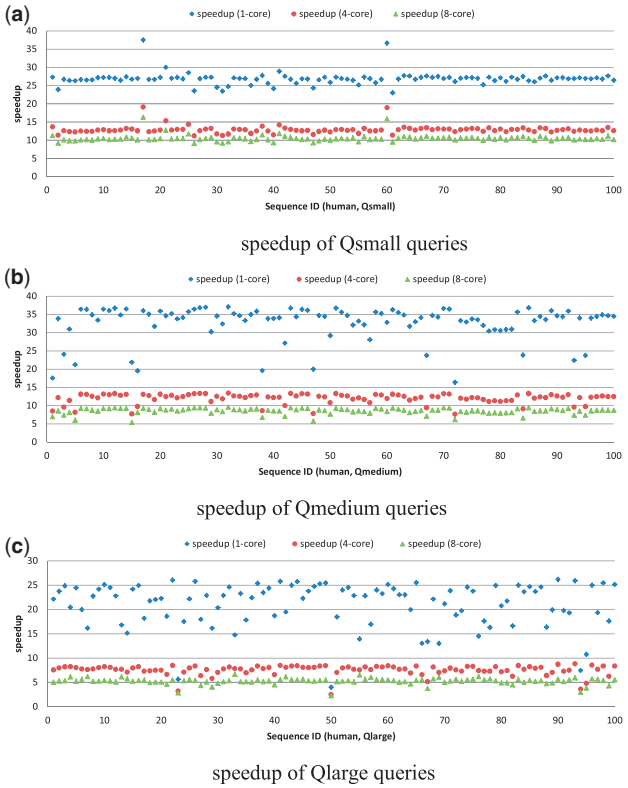

Fig. 12. Speedup of G-BLASTN on human genome database (GTX780 versus Two Xeon E5620)

4.2 Experimental results

4.2.1 Performance under normal mode Under normal mode, G-BLASTN handles the queries one at a time. We first present the experimental results of 'megablast'. The speedups over 8-core platform on human genome database are shown in Figure 12. The speedups of other experiments are shown in Supplementary Figures S3–S6. We also show the average speedup of each query set in Table 2. The overall speedups are calculated as the average of all 1600 query experiments for each hardware setting. As

Table 2. Average speedup of G-BLASTN under 'megablast' mode

Database	Query	Intel i7-3820		Intel Xeon E5620		
		1-core	4-core	1-core	4-core	8-core
Human	Qsmall	10.47	5.11	26.90	12.88	10.52
	Qmedium	11.49	4.53	32.68	11.98	8.50
	Qlarge	9.22	3.37	21.07	7.54	5.25
	Qbac	10.80	5.37	26.04	12.71	10.47
Mouse	Qsmall	18.50	9.37	44.12	21.87	18.28
	Qmedium	17.84	7.39	49.16	19.11	14.32
	Qlarge	10.44	4.14	23.16	9.14	6.92
	Qbac	20.97	10.73	49.28	24.71	20.80
Overall		**14.80**	**7.15**	**35.85**	**16.85**	**13.76**

The meaning of the "overall" values are the average speedups of all query experiments for each hardware setting.

Table 3. Speedup of G-BLASTN under 'blastn' mode using Qbac query set

Database	Intel i7-3820		Intel Xeon E5620		
	1-core	4-core	1-core	4-core	8-core
Human	4.58	1.57	8.01	2.99	2.15
Mouse	5.02	1.83	8.84	3.51	2.70
NCBI nt	3.37	1.29	5.71	2.36	1.74
Overall	**4.32**	**1.56**	**7.52**	**2.95**	**2.20**

The meaning of the "overall" values are the average speedups of all query experiments for each hardware setting.

compared with 4-core Intel i7-3820, G-BLASTN achieves an overall speedup of 7.15X. As compared with the 8-core platform, G-BLASTN achieves an overall speedup of 13.76X. There are several reasons why BLASTN runs much faster on i7-3820 than on Xeon E5620. First, i7-3820 has a much higher working frequency than E5620. Secondly, the memory bandwidth of i7-3820 is twice of E5620. Thirdly, the memory module of our i7-3820 platform is faster than that of E5620 platform. Based on the results of E5620, we can notice that the speedups achieved using 8 cores are only slightly better than using 4 cores.

We also notice that the speedups on the human database are less than those on the mouse database. This is mainly because the human build 36 database consists of 367 sequences, >200 of which are short sequences (<1 million bp). In contrast, the mouse build 36 database consists of 21 very long sequences. We will discuss more on this issue in Section 5.

We evaluate the performance of 'blastn' mode using Qbac query set against human, mouse and the partial NCBI nt databases. We show the average speedups of each database and the overall speedups in Table 3. G-BLASTN achieves an overall speedup of 1.56 and 2.95 as compared with 4-core i7-3820 and 8-core E5620, respectively. As compared with 'megablast' mode, 'blastn' mode has a much smaller value of stride size, which results in more scanning workload and more seed hits. Therefore the ungapped extension step under 'blastn' mode takes much longer time than under 'megablast' mode, as shown in Supplementary Figure S7. Since the ungapped extension is sequentially executed on CPU, the speedups achieves under 'blastn' mode are much less than 'megablast'.

Table 4. Speedup of pipelined G-BLASTN (GTX780 versus Intel i7-3820)

Mode	Database	Query	1-core	4-core
megablast	Human	Qsmall	10.83	5.28
		Qmedium	12.67	5.05
		Qlarge	12.19	4.68
	Mouse	Qsmall	20.49	10.37
		Qmedium	20.12	8.51
		Qlarge	12.09	5.47
blastn	Human	Qbac	5.49	1.87
	Mouse	Qbac	6.49	2.36
	NCBI nt	Qbac	4.73	1.86

4.2.2 Performance under pipeline mode To evaluate the performance of pipelined G-BLASTN, we use all queries in each dataset as a single input to G-BLASTN. The speedups against the NCBI BLAST on Intel i7-3820 are shown in Table 4. If we compare Table 4 with Tables 2 and 3, we can observe a significant improvement on the speedups for many datasets. For small and medium queries under 'megablast', the trace-back and output steps account for a very small portion of the total time and hence the pipeline design does not offer much of an advantage. For large queries using 'megablast' in which the trace-back and output steps take a much longer time, the pipeline design hides a significant portion of time. Under 'blastn' mode, the pipeline design can further improve the speedups of the Qbac query set by 19–44%.

5 DISCUSSIONS AND CONCLUSIONS

In this article, we describe our design and implementation of G-BLASTN, an open source software tool for nucleotide alignment based on the widely used NCBI-BLAST. G-BLASTN exploits the power of GPUs to accelerate nucleotide alignments. Compared with a contemporary quad-core Intel CPU running at 3.6 GHz, G-BLASTN on a single $650 GPU card can achieve overall speedups of 14.8X and 4.32X under 'megablast' mode and 'blastn' mode, respectively. When compared with multithreaded NCBI-BLAST that uses four CPU cores, G-BLASTN can still achieve overall speedups of 7.15X ('megablast') and 1.56X ('blastn'). G-BLASTN also supports a pipeline mode that further improves the overall performance by up to 44% when handling multiple queries.

G-BLASTN can be improved in the following directions. At present, G-BLASTN invokes a kernel function for each database sequence, which is not efficient when the length of the database sequence is shorter than one million bps. There are several possible solutions to this problem. One possibility is to aggregate short database sequences into longer ones. Another solution is to process multiple database sequences in each kernel function call. G-BLASTN is also limited by the GPU memory size. We first

plan to extend G-BLASTN to support multiple GPU cards. Doing so can not only support larger databases, but also achieve better speedups. A long-term solution is to divide the huge database into smaller volumes, and then scan each volume in GPU one by one. To minimize the overhead of copying database into GPU memory, multiple queries should be processed at a time. A more challenging task is to accelerate other steps such as ungapped extension, gapped extension and trace-back, which will improve the performance of 'blastn' mode significantly. Finally, we also plan to support 'discontiguous megablast mode' in our future work.

Funding: Hong Kong Baptist University (grant FRG2/11-12/158).

Conflict of Interest: none declared.

REFERENCES

Altschul,S.F. *et al.* (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Altschul,S.F. *et al.* (1997) Gapped BLAST and PSI-BLAST:Anew generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Camacho,C. *et al.* (2009) BLAST+: architecture and applications. *BMC Bioinform.*, **10**, 421.

Dematte,L. and Prandi,D. (2010) GPU computing for systems biology. *Brief. Bioinform.*, **11**, 323–333.

Fei,X. *et al.* (2008) FPGA-based accelerators for BLAST families with multi-seeds detection and parallel extension. In: *Proceedings of the 2nd International Conference in Bioinformatics and Biomedical Engineering*. IEEE, Shanghai, China, pp. 58–62.

Jacob,A. *et al.* (2007) FPGA-accelerated seed generation in Mercury BLASTP. In: *Proceedings of 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, California, USA, pp. 95–106.

Lin,H. *et al.* (2008) Massively parallel genomic sequence search on the Blue Gene/P architecture. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. ACM/IEEE, Austin, USA, pp. 1–11.

Ling,C. and Benkrid,K. (2010) Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm. *Proc. Comput. Sci. USA*, **1**, 495–504.

Liu,C.M. *et al.* (2012a) SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, **28**, 878–879.

Liu,Y. *et al.* (2012b) CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics*, **28**, 1830–1837.

Lu,M. *et al.* (2013) GPU-accelerated bidirected De Bruijn graph construction for genome assembly. *Web Tech. Appl. Lect. Notes Comput. Sci.*, **7808**, 51–62.

Lu,M. *et al.* (2012) High-performance short sequence alignment with GPU acceleration. *Distrib. Parallel Dat.*, **30**, 385–399.

Manavski,S. and Valle,G. (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinform.*, **9** (**Suppl. 2**), S10.

Morgulis,A. *et al.* (2006a) WindowMasker: window-based masker for sequenced genomes. *Bioinformatics*, **22**, 134–141.

Morgulis,A. *et al.* (2006b) A fast and symmetric DUST implementation to mask lowcomplexity DNA sequences. *J. Comp. Biol.*, **13**, 1028–1040.

Morgulis,A. *et al.* (2008) Database indexing for production MegaBLAST searches. *Bioinformatics*, **24(16)**, 1757–1764.

Nguyen,V.H. and Lavenier,D. (2009) PLAST: parallel local alignment search tool for database comparison. *BMC Bioinform.*, **10**, 329.

Nickolls,J. (2007) Nvidia GPU parallel computing architecture. In: *Proceedings of the IEEE Hot Chips 19*. IEEE, Stanford, CA, USA.

Owens,J.D. *et al.* (2008) GPU Computing. *IEEE Proc.*, **96**, 879–899.

Sotiriades,E. and Dollas,A. (2007) A general reconfigurable architecture for the BLAST algorithm. *J. VLSI Signal Process.*, **48**, 189–200.

Vouzis,P.D. and Sahinidis,N.V. (2011) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, **27**, 182–188.

Zhang,Z. *et al.* (2000) A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.*, **7**, 203–214.