An Analysis of BLASTP Implementation on NVIDIA GPUs

David Glasco 6/8/2012

1.0 Introduction

Given a new protein sequence, scanning existing protein databases for similar sequences is becoming an important and repeated task in bioinformics. Initially, the Smith-Waterman [1] algorithm was developed to find local, optimal matches between an input sequence and a given database of existing protein sequences. The Smith-Waterman algorithm uses a recursive algorithm (dynamic programming) to perform the search. Unfortunately, the algorithm is too compute intensive and has an execution time that is quadratic with respect to the length of protein sequences [2].

As the sequencing rate for new proteins increased, the need for a faster alignment tool became apparent, and a heuristic-based algorithm was a potential direction. The BLAST algorithm is such a heuristic-based algorithm, and it is based on the assumption that good alignments contain short, high scoring matches. In addition to more efficient algorithms, more efficient computational devices are needed as traditional CPUs are reaching their limits of performance and power scaling, while GPUs continue to offer a significant and increasing peak GLOPs and GLOPS/Watt as compared to CPUs. This paper analyzes several implementations of the BLASTP algorithm. The goal is to understand the current algorithmic bottlenecks on the GPU and then to briefly examine potential areas of improvement.

This paper is organized as follows. Section 2 gives an overview of the BLASTP algorithm, and section 3 briefly describes the NVIDIA GPUs that are used to implement the BLASTP algorithm. Next, section 4 describes three recent GPU implementation of the BLAST algorithm, and section 5 does a deep dive into one of the three algorithms. Then, given this background knowledge, section 6 examines the performance of this algorithm on a current NVIDIA GPU (GTX 680). Finally, section 7 will provide some thoughts on potential improvements, and section 8 will conclude.

2.0 BLASTP

As noted above, the Basic Local Alignment Searching Tools for Proteins (BLASTP) is an algorithm that is based on the observation that good alignments typically have short matches, but given that the algorithm is an approximation (heuristic) is it possible for it to miss optimal alignments that algorithms such as Smith-Waterman would have identified. The BLAST algorithm has four stages [3]:

1) Hit (word match) Detection
2) Ungapped Extension
3) Gapped Alignment
4) Gapped Alignment with traceback

The figure below gives an example of the first three stages [3].

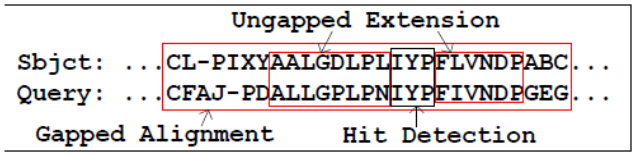

Figure 1: BLASTP Example

In this example, the query sequence is compared to the subject sequence. In stage 1 (Hit Detection), the exact match of the sequence "IYP" is identified. In stage 2, ungapped extension is performed on pairs of high-scoring segment pairs (HSPs). In this example, the ungapped extension extends the initial match in both directions. In stage 3, the ungapped extension is extended using a gapped alignment. Both stage 2 and stage 3 use a scoring matrix and threshold to determine the extent of the alignment. In the final stage, a traceback algorithm is used to generate and score the alignments.

The figure below provides more details of the process [4]. In stage 1, short, matches are identified (black lines in the left figure). In stage 2, matches along the same diagonal are extended (non-gapped) if the resulting score exceeds a specified threshold. The extensions are shown as grey lines in the left figure. Next, stage 3 extends (typically using Smith-Waterman) the non-gapped sequences using gapped alignment, as shown by the grey line in the right figure. Finally, stage 4 generates and scores the sequence for the end user using alignment traceback algorithms.

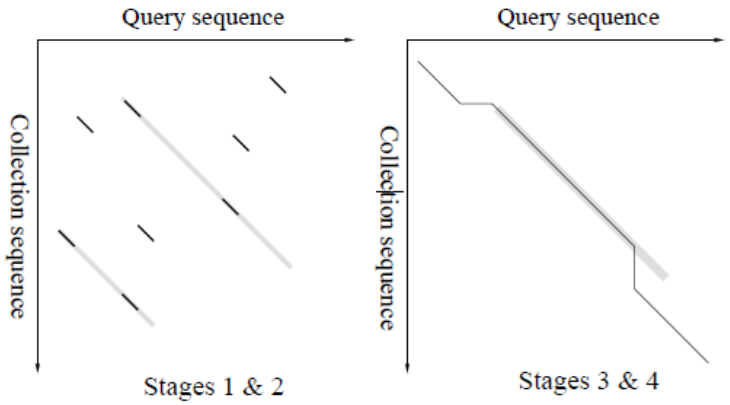

Figure 2: BLASTP Example

3.0 GPU Architecture

3.1 Architecture Overview

As noted above, GPUs offer a potential performance advantage over existing CPUs, as they offer higher peak GFLOPS and are more power efficient (GLOPS/Watt). The NVIDIA GPU consists of a number of Streaming Multiprocessors (SM), of which pairs are grouped into a GPC block, as shown in the figure below. The four GPC blocks are connected to a distributed, 512KB L2 cache, and the L2 cache is

connected to four memory controllers. The green blocks in the diagram represent the individual computation engines within the SMs.

Details of the architecture can be found in the GK104 white paper [5].

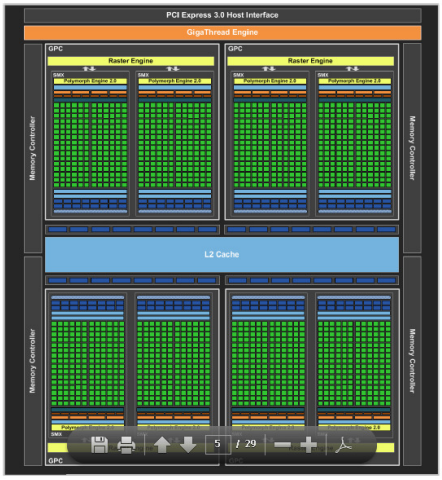

Figure 3: NVIDIA GTX 680 GPU

3.2 Performance Characteristics

Within each SM, warps of 32 threads are executed using a Single Instruction, Multiple Thread (SIMT) model. In this model, peak performance is obtained when both the control flow and the memory accesses are converged. That is, when the threads within a warp execute the same instruction and generate well behaved memory accesses (same word across the thread or adjacent words, for example) peak performance is obtained. If either the control or memory streams diverge, then performance is diminished. Management of this divergence along with data placement is critical to performance, but understanding divergence is a difficult problem and is for the most part ignored in the existing literature analysis of CUDA implementations of BLASTP.

In addition to managing the divergence, data placement is also critical to performance. Within a CUDA application, data may be placed in global memory, shared-memory, constant memory or texture memory. Global memory is persistent and accessible to all threads of a given application. Shared-memory is accessible to threads within a thread block and has a lifetime of the threads within the block. Shared memory is a local, high-bandwidth and low latency memory store. Constant memory and texture memory are read-only memory structures that provide cached data access. The data bandwidth of the texture memory tends to be much higher than that of global memory or constant memory. The performance section of this paper will examine how data placement can impact performance.

4.0 GPU Approaches

This investigation examined three GPU implementation of BLASTP:

1) GPU-NCBI-BLAST [6]

2) CUDA-BLASTP [2]
3) GPU-BLASTP [3]

As highlighted in all three implementation, the data structure used to enable the initial short word match (essentially an optimized lookup table) is critical to performance. Also, each implementation uses the same methodology to presort the database by subject sequence length, which allows blocks of similar length subject sequences to be concurrently scanned by the GPU. The goal of the sort is to minimize difference in the execution time of the concurrent GPU threads.

Section 4 and 5 will focus on the characteristics of CUDA-BLASTP. The analysis will examine the key characteristics of the implementations, as described in the respective papers. Comparing the performance data from the papers is difficult, as the system configurations, subject sequences and query sequences are different in each case. The source code of GPU-BLASTP, which is the most recent and highest performing implementation, was not publically available.

4.1 GPU-NCBI-BLAST

One goal of the GPU NCBI-BLAST implementation is that the results of the algorithm exactly match those produced by the NCBI-BLAST tool[1], and as a result of this goal, the GPU implementation is built on top of the NCBI-BLAST code base. The implementation focuses on the first two steps of the BLAST algorithm, as the authors observe that these two steps consume over 75% of the execution time, and the authors claim a 4x speedup from their GPU implementation over a single-thread CPU implementation of NCBI-BLAST. [6]

The algorithm preprocesses the query sequence and generates a query-index table – a form of lookup table. The table stores how many time each word is found in the query and the location of the word in the sequence. The query-index table holds the location of the first three occurrences of a word and has the ability to overflow into secondary structure if more occurrences are found. This query-index table is stored in global memory due to its size, but the information that specifies if a word is found in the query sequence (the presence bits) is extracted from the table and stored in shared-memory. The query sequence is stored in constant memory, and the database sequences are stored in global memory. The figure below illustrates this organization [6].

The detailed organization of the data structures and how they are walked by the algorithm is not described in the paper, but the presences bits and the query-index table are likely indexed by the w-mer that is currently being scanned.

[1] How important is it for the implementation to match the NCBI-BLASTP results?

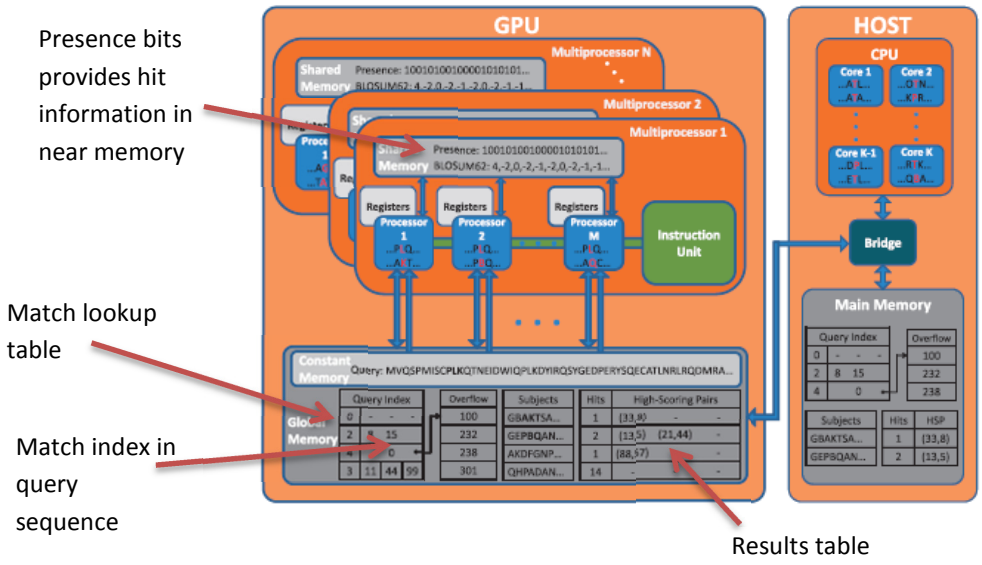

Figure 4: GPU-BLAST Data Organization

The algorithm proceeds as follows. First, each thread scan consecutive words of a subject sequence and checks the presence bits to determine if a match is found. Given the presence bits are stored in shared-memory, this operation is very efficient. For matches, ungapped extensions are performed using a substitution matrix stored in shared-memory and a scoring threshold. The CPU, in parallel, to the GPU also processes a set of subject sequences rather than wait for the GPU to complete. The gapped extension and traceback steps are performed on the CPU using the existing NCBI-BLAST implementation.

4.2 CUDA-BLASTP (Liu)

As with GPU-NCBI-BLAST, one of the key focus areas for this algorithm is data structure design. In this implementation, a compressed deterministic finite state automaton (DFA) is used to store the word match information for the query sequence, and this query sequence information is preprocessed on the CPU. The DFA structure is essentially an optimized table lookup structure that for a given state (based on a pair of sequence characters) returns the next state, a pointer to a secondary structure that provides word hit information (hit or miss) and pointers to the location in the sequence for the word. The DFA is similar, but not identical to the DFA structure presented in FSA-BLAST [4]. Figure 5 below shows the data structures.

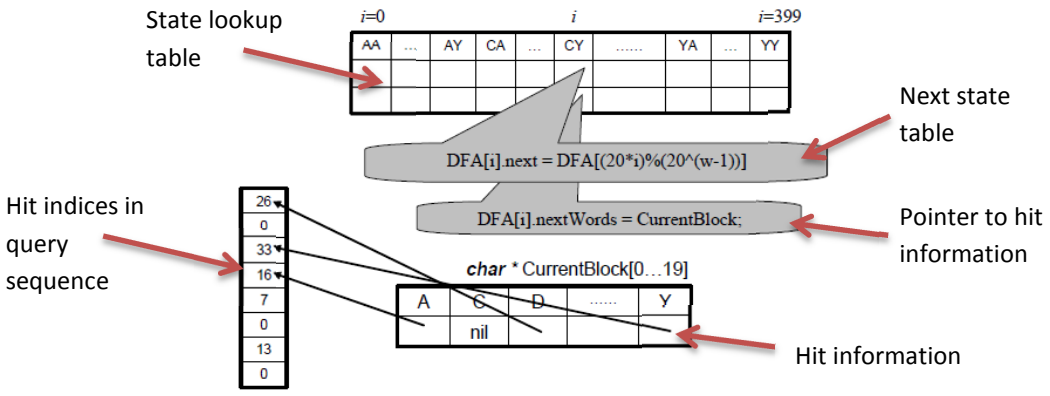

Figure 5: CUDA-BLASTP DFA

In a simple word lookup table, each w-mer (W = 3 in this investigation) is used to index into a large lookup table. With an alphabet of length A, the table size would be A^3. The CUDA-BLASTP optimizes this structure by only storing a prefix of length w-1. The DFA[i].nextWord pointer points to an additional vector of length A that used to hold hit information and subsequently points to sequence locations. These optimizations can significantly reduce the size of the table when the table is sparse.

The DFA next state information is stored in constant memory, and the remaining portions of the DFA and the subject sequences are stored in texture memory.

Stages 1-3 of the BLASTP algorithm are implemented on the GPU, and stage 4 is implemented on the CPU. As with the previous algorithm, the subject sequences from the database are sorted, and batches of similar length sequences are processed on the GPU sequentially. Stages 1-2 are combined into a single kernel invocation, and stage 3 implements the Smith-Waterman alignment algorithm using a second kernel on the GPU.

The flow of the algorithm is as follows. Each thread reads a subject sequence and uses the DFA to identify word hits. For each hit, ungapped extension is performed if an additional match has already been identified along the diagonal and is within a specified threshold distance. After all subject sequences are processed, the hit information is returned to the CPU for post processing. Afterwards, the information is returned to the GPU to execute stage 3 – gapped extension using Smith-Waterman. Finally, the results are returned to the CPU for stage 4 calculations (traceback) and final output formatting.

As for performance, the authors claim roughly a 10x speedup over a NCBI-BLAST implementation on a single CPU thread, but this algorithm does not guarantee that its results will match that of an identical search performed by NCBI-BLAST. Section 4 of this paper will examine the performance of this algorithm in more depth.

4.3 GPU-BLASTP (Xiao)

The third GPU implementation of BLAST is GPU-BLAST [3]. As with CUDA-BLAST, the first two stages of the BLASTP algorithm are combined into a single kernel, and the flow of the algorithm is almost identical to that used in CUDA-BLAST. Although the authors did explore two additional ideas – two-level results buffering and load balancing. In CUDA-BLAST a fixed sized results buffer is used, while GPU-BLAST implements a two-level buffer scheme in which each thread is given a small, fixed sized results buffer and a mechanism to spill over into a common shared buffer. For load balancing, GPU-BLASTP uses a work queue structure to allow threads to obtain the next subject sequence to scan. A DFA, which is also based on the FSA-BLAST [4] scheme, is used to hold the word lookup table.

In addition to examining load balancing, this work also examinations the performance impact of using different GPU memory types to hold the data structures. For the memory types, the paper examines both constant memory and texture memory to hold subject sequences, word lookup tables and scoring matrices. The authors found that the best performance for stage 1 and 2 is typically obtained when the subject sequences and the word lookup table are stored in texture memory.

4.4 Summary

Overall, the three implementations are very similar. They all focus on implementing stages 1 & 2 on the GPU, as these stages are the bulk of the execution time for the algorithm. All three approaches focus the majority of the effort on data structure design and placement. Unfortunately, the performance numbers are difficult to compare as the system configurations and test sequences are not identical, but a rough estimation of performance would indicate that GPU-BLASTP is 2x faster than CUDA-BLASTP, and that CUDA-BLASTP is roughly 10x faster than GPU-NCBI-BLASTP. The next section will provide a detailed description of the CUDA-BLASTP implementation, and section 6 will examine the resulting performance under a number of configurations.

5.0 GPU Implementation of CUDA-BLASTP

As noted above, the CUDA-BLASTP implementation consists of four stages of which stage 1 and stage 2 are combined into a single CUDA kernel (`Blast_Kernel`), stage 3 is implemented with a single CUDA kernel (`sub_based_SW`) and stage four is performance on the CPU. In addition, the database is initially sorted by sequence length to improve GPU load balance, and then divided into smaller blocks, as shown in the figure below. The goal of the sorting is to attempt to minimize the variations in thread execution time, although execution time is dependent on both the subject sequence length and the number of hits identified and extended.

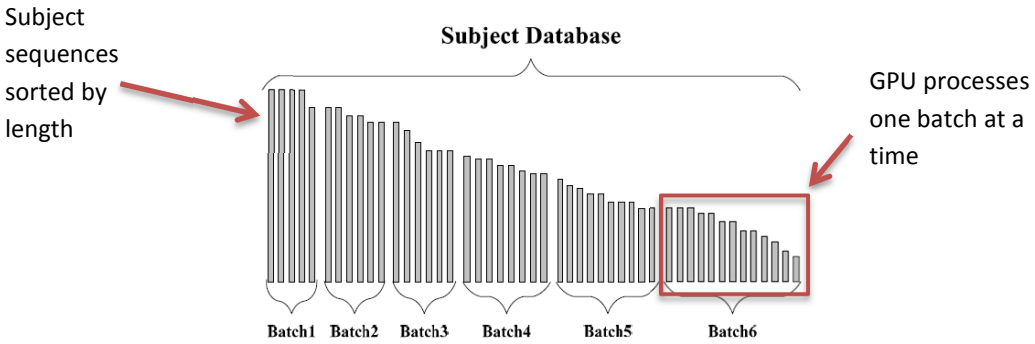

Figure 6: Database Sorting and Batching

5.1 Execution Timeline

The figure below shows an example execution timeline for the CUDA-BLASTP implementation. The three rows of interest in the figure are MemCpy (HtoD), MemCpy (DtoH) and Compute. The MemCpy blocks are the memory copies from Host (CPU) to Device (GPU) or Device to Host. The Compute row represents the kernel execution. Initially, the DFA structure and scoring matrix are copied to the GPU, and then a batch of the subject sequences are copies to the GPU. Next, the GPU executes stage 1 and 2, and the results are copied back to the CPU. Finally, the CPU post-processes the resulting hit data and copies the high scoring hits back to the GPU where the GPU executes the second kernel, which uses the Smith-Waterman algorithm to perform gapped alignment.

As shown in the timeline, the data copies consume a large fraction of the execution time compared to the blast kernel (stage 1 and 2). The stage 3 (Smith-Waterman gapped alignment) takes a negligible amount of time. The figure highlights the processing of the second batch of subject sequences.

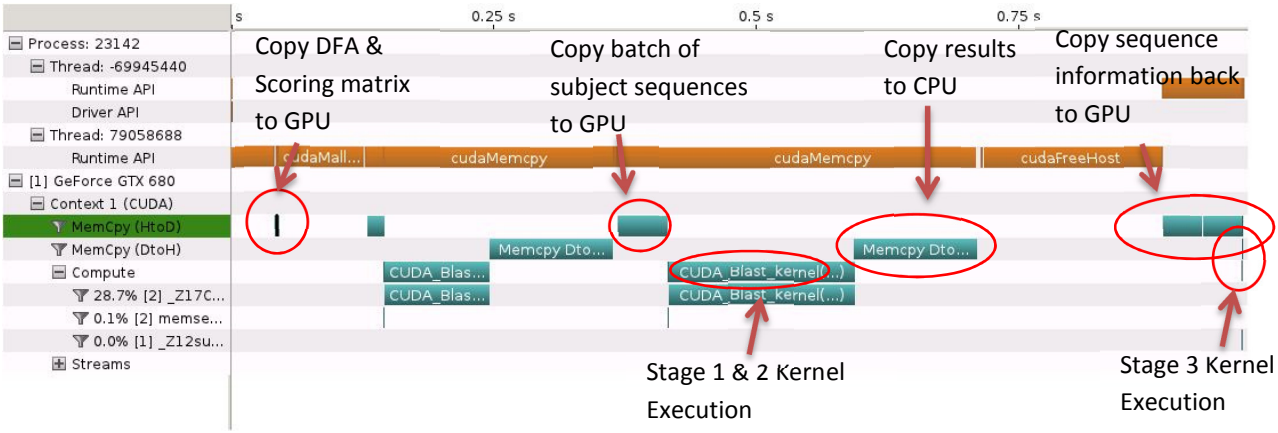

Figure 7: Example CUDA-BLASTP Execution Timeline

5.2 Data Structures

CUDA-BLASTP has several key data structures, as shown in the table below. The first, and most important, is the DFA implementation of the word lookup table, which was shown in Figure 5. As shown in the table below, the actual implementation of the DFA is split into two components – the next

state information and the next word information. In all experiments, an alphabet size of 20 and a word length of 3 are used.

Structure	Information Held	Organization	Location
DFA[i].next	Next state for the DFA	Vector[20 * 20]	Constant Memory
DFA[state][i].nextWord	Pointer to hit information	2D Array[20 * 20][20]	Texture Memory
CurrentBlock[state]	Hit information and pointer to sequence indexes	2D Array[20 * 20][20]	Texture Memory
querySequences	Sequence indexes	Vector[variable]	Texture Memory

Table 1: CUDA-BLASTP Data Structures

The DFA is used as follows. First, the next state and next word entries are read using the current state (state) and the current subject sequence character (i). If the next word entry is non-zero, then a hit has been found. Otherwise, the state moves to the next state, and the process repeats until the input subject sequence has been exhausted. For a hit, the current block and query sequence information is read. This information is used to determine if an ungapped extension should be performed.

5.3 GPU Kernels

As noted above, the implementation uses two CUDA kernels - the first to perform stage 1 and 2 of the BLASTP algorithm (hit detection and ungapped extension), and the second kernel to perform stage 3 (gapped extension). As suggested above and will be quantified below, the second kernel consumes a negligible amount of time; and therefore, the focus of the analysis will be on the first kernel (`Blast_Kernel`).

The `Blast_Kernel` consists of several phases, as listed below. Initially, the kernel executions a small amount of initialization code (Kernel Initialization). Next, the kernel loops through each sequence assigned to the thread (Sequence Processing). As part of this sequence processing, the kernel performs a small amount of initialization work (Sequence Initialization), and then it loops through the subject sequences (Sequence Scanning) looking for hits (**Hit Check**). If a hit is identified, it is processed (Hit Processing) and potentially extended (Hit Extend).

1. Kernel initialization
2. Sequence Processing
   2.1. Sequence Initialization
   2.2. Sequence Scanning
      2.2.1. Hit Check
         2.2.1.1. Hit Processing
            2.2.1.1.1. Hit Extend

The majority of the execution time will be shown to be in the **Hit Check** portion of the `Blast_Kernel`.

6.0 Performance Analysis

6.1 Protein Sequences and Databases

To examine the performance of this implementation, five query sequences of various lengths are used, and the table below provides the length information for the sequences[2].

Sequence	Length
P14144	127
P42018	254
Q52TG9	517
Q52KR2	1054
P08678	2026

Table 2: Query Sequence Lengths

Both the swissprot and the env_nr protein sequence databases from NCBI were used as the scanned database, although the majority of the performance results will use the larger env_nr database.

The table below shows the BLASTP parameters used in the study. These values are the default values for CUDA-BLASTP.

Parameter Description	Value
Word Size	3
Drop-off value for ungapped extension	7
Drop-off value for gapped extension	15
Drop-off value for triggering gaps	22
Drop-off value for final gaped extension	25
Open Gap Penalty	7
Extension Gap Penalty	1

Table 3: BLASTP Parameters

6.2 GPU Specification

The GPU used was a GTX680, which has the characteristics shown in the table below.

GTX680	
Number SM	8
Warp Size	32 threads
Threads per SM	2048
Total L2 Size	512 KB

Table 4: GTX 680 Specifications

6.3 Initial Results

As noted above, the execution consists of two basic components – CUDA kernel execution and data movement (MemCpy) – in addition to the CPU processing. The table below shows the relative percentage of execution time that each of these components consumes across the 5 query sequences

[2] These are the same 5 sequences used by Liu in the CUDA-BLASTP evaluation.

under study. The first table shows the result when scanning the smaller, swissprot database, and the second table shows the results when scanning the env_nr database.

swissprot Database	P14144	P42018	Q52TG9	Q52KR2	P08678	Ave
MemCpy HtoD	21.4%	19.7%	18.7%	14.3%	11.5%	17.1%
MemCpy DtoH	36.0%	35.7%	32.8%	25.8%	20.7%	30.2%
CUDA_Blast_kernel (...)	42.3%	44.4%	47.7%	57.8%	66.7%	51.8%
sub_based_SW (...)	0.1%	0.0%	0.6%	1.9%	0.9%	0.7%

Table 5: CUDA-BLAST Execution Time Breakdown Scanning swissprot Database

env_nr Database	P14144	P42018	Q52TG9	Q52KR2	P08678	Ave
MemCpy HtoD	15.2%	14.9%		11.3%	11.5%	13.2%
MemCpy DtoH	20.8%	20.3%		13.4%	20.7%	18.8%
CUDA_Blast_kernel (...)	63.9%	64.8%		71.8%	66.7%	66.8%
sub_based_SW (...)	0.0%	0.0%		3.5%	0.9%	1.1%

Table 6: CUDA-BLAST Execution Timeline Breakdown Scanning nr Database

The Q52TG9 query sequence caused the program to crash for unknown reasons when using the env_nr database.

As shown in the two tables above, the CUDA `Blast_Kernel` consumes the majority of the execution time, and the sub_based_SW kernel consumes a negligible amount of time. For the smaller swissprot database in table 5, the data movement consumed a significant amount of the execution time (47.3%) compared to the computation phases (52.5%), but as the size of the data base increases, the overhead decreases as shown in Table 6 for the env_nr database – 32% data movement vs. 67.9% computation. As noted above, the MemCpy HtoD is mainly the copy of the database sequences to the GPU, and the MemCpy DtoH is the copy of the results from the GPU to CPU. For this initial investigation, the focus will be on the analysis of the CUDA `Blast_Kernel`, but the final section of the paper will discuss possible future work to address the data movement overhead.

6.4 Blast_Kernel Execution.

As described above, the CUDA `Blast_Kernel` combines both stage 1 (hit detection) and stage 2 (ungapped extension) of the BLASTP algorithm. The table below shows the basic statistics for stage 1 and 2 for a single thread of the execution when the env_nr data based is scanned for the five query sequences under study. As noted above, the Q52TG9 scan terminated prematurely, but the relative statistics are likely still valid.

env_nr	P14144	P42018	Q52TG9	Q52KR2	P08678
Sequences	2961	2961	1918	2961	2961
Bases	1132719	1132719	1006105	1132719	1132719
Hits	82167	118555	246832	449644	647022
Extensions	1442	1365	6708	9971	18780

Table 7: Blast_Kernel Computation Breakdown

As expected, the number of hits and extensions tends to grow with the length of the query sequence. The following table shows a very rough breakdown of the execution time of the `Blast_Kernel`.[3] The first two rows show the percentage of subject sequences that resulted in a hit and an ungapped extension. The subsequent rows show the execution time breakdown of the components of `Blast_Kernel`, as described in Section 5.3.

nr	P14144	P42018	Q52TG9	Q52KR2	P08678	Ave
Hits (%)	7%	10%	25%	40%	57%	28%
Extensions (%)	0%	0%	1%	1%	2%	1%
Kernel Init Time	0%	0%	0%	0%	0%	0%
Sequence Init Time	3%	3%	2%	2%	1%	2%
Hit Check Time	**93%**	**92%**	**82%**	**68%**	**51%**	**77%**
Hit Process Time	4%	5%	15%	29%	46%	20%
Hit Extend Time	0%	0%	1%	1%	2%	1%

Table 8: Blast_Kernel Execution Time Breakdown

As shown in the table, the **Hit Check** processing portion of the kernel consumes the majority of the execution time, and the hit processing time grows as the number of hits in a given subject sequence increases. The execution time of the kernel initialization (Kernel Init), the subject sequence initialization (Sequence Init) and the ungapped extension time (Hit Extend) are negligible and will not be examined in this work.

The **Hit Check** step consists of reading the next character in the subject sequence, determining if the end of the subject sequence has been reached, and reading the *next state* and *next word* information from the DFA data structure. If a hit is found, then the hit process step consists of reading the *current block* and *sequence offset* information and determining if a previous hit on the same diagonal is within a distance threshold.

The work performed by the HitCheck phase of the kernel is shown in pseudo-code below.

```
address = starting location of sequence in database
while (! End of sequence) {
        letter = Tex_Read_Subject[address];
        // currentWord = DFA[state][letter].nextWord
        currentWord = Tex2D_Read_DFA[letter][currentGroup].nextWord
        // nextGroup = DFA[state + letter]
        nextGroup = Constant_Read_DFA[currentGroup + letter].nextGroup
        if (currentWord)
                Process/Extend Hit
}
```

The loop consists of two texture reads, a constant read and the necessary address calculations. Section 7 will examine alternative data placement for these two structures. The key to improving performance

[3] The NVIDIA performance analysis tools only provide a way to measure total wall clock time of various segments of a running kernel, and given that multiple threads are time slicing the SM execution units, the runtimes are pessimistic. In this analysis, I assume that the time slicing interference is roughly equal across all threads and therefore, the percentage breakdown of a kernel execution time is roughly correct.

of this algorithm on GPUs is to optimize this step of the algorithm and the data movement phases, as mentioned above.

7.0 Areas for Improvement

Improving the execution time of the BLASTP algorithm on GPUs potentially requires improvements in three areas: data structure and placement, data movement and control/data divergence. This analysis will focus on the data placement question. Ideas to improve data movement and divergence control will be discussed in Section 8, but they are beyond the scope of this initial study.

7.1 Data Placement

As observed in several of the previous sections, optimizing the data structures and placement is key to good GPU performance. In the current implementation of CUDA-BLASTP, the key data structures are mainly stored in texture memory with the exception of the DFA[i].nextGroup vector which is stored in constant memory. This section will look at opportunities for optimizing the placement of these structures.

7.1.1 nextGroup

As shown in Figure 5, the DFA nextGroup pointer can be deterministically calculated based on the current subject sequence and the word length.

$$DFA[i].next = DFA[(20 * i) \% (20^{(w-1)})] \text{ where } i = subject\_sequence_i$$

Hence, the lookup into a pre-calculated table could be replaced with a local calculation rather than a lookup of constant memory.

Unfortunately, the change has little impact on the execution times, as shown in the table below. The table shows the relative execution time of the optimized version vs. the base version (Opt/Base * 100%).

	P14144	P42018	Q52TG9	Q52KR2	P08678	Ave
Calculate nextGroup	99%	99%	100%	103%	98%	100%

Table 9: Optimizations: Calculate nextGroup

The overall cost of a cached constant memory lookup is not significantly more than the calculation above, especially given the modulo 400 operation in the equation. Replacing the modulo 400 by a power of two shift either by padding the vector or reducing the alphabet to 16 might improve performance, but still it is unlikely.

7.1.2 Shared Memory

Another optimization is to move the DFA[i].nextWord from a 2D texture memory to the local Shared Memory. The shared memory block is a high bandwidth, low latency per SM R/W memory structure. The one disadvantage of shared memory is that the application must explicitly load the shared memory prior to its use. Another disadvantage of using Shared Memory is that the Shared Memory space (32KB per SM) must be shared across all active blocks on a given SM.

In this application, the 2D array, which is a 400x20x2 (16000) byte structure, must be copied into each SM (8 total) for each block (96 in the current configuration) executed, as the lifetime of the shared memory data is the block execution lifetime. In addition, this 16000 byte shared memory allocation limits the number of active blocks that can run on the SM to 2, as the 32KB shared memory block must hold this allocation (16000) for all active blocks. Table 10 below shows the resulting performance for this configuration. As can be seen, the limited number of active blocks minimizes the potential gains from using shared memory.

	P14144	P42018	Q52KR2	P08678	Ave
Shared Memory	128%	131%	155%	167%	145%

Table 10: Optimizations - nextWord in shared memory

The shared memory implementation has on average a 45% longer execution time. Although, per kernel statistics shows that when using shared memory, each threads runs in roughly 70% of the time as compared to the base implementation. Hence, with a shared memory implementation, each thread runs in approximately ¾ of the time compared to the base configuration, but due to size limitations, only half as many threads can be running at any given time.

If storage for the alphabet could be compressed or reduced, then more DFA structures could be held in shared memory. For example, if the 1600B footprint could be reduced to 8192B, then the shared memory on a given SM could hold twice the DFA structures. This would allow twice the number of thread blocks to execute on a given SM, unless other resources limited the number of blocks or threads that could be concurrently executed. In addition, if the DFA state could be compressed to 8b rather than 10b, then potentially more thread blocks could run concurrently.

7.1.3 Constant Memory

The final experiment is to move the nextWord structure into constant memory. The table below shows relative execution times when constant memory is used for this structure rather than texture memory; the average slowdown is 2%. As with the previous two optimizations, the results show that texture memory continues to be the most efficient memory.

	P14144	P42018	Q52KR2	P08678	Ave
Constant Memory	106%	105%	102%	97%	102%

Table 11: Optimizations - nextWord in constant memory

Overall, these studies have demonstrated that the texture memory is the most efficient memory for these types of read-only, sparsely accessed data. This result corroborates the similar conclusion reached in the GPU-BLASTP work [3].

8.0 Conclusions

8.1 Initial Analysis

This paper examined the detailed implementation and execution behavior of the CUDA-BLASTP implementation. The analysis shows that roughly 50-70% of the overall execution time is spent in the two CUDA Kernels. The remaining time is spent moving data between the GPU and CPU memory systems. In addition, the analysis shows that roughly 77% of the kernel execution time is spent in the **Hit Check** portion of the kernel, in which the DFA structure is accessed to determine if the current subject sequence word has a match in the query sequence. The analysis focused on optimizing the lookups into this DFA structure.

To address the performance of the **Hit Check** functionality, three alternative schemes were examined. First, the DFA.nextGroup texture access was replaced with a direct computation. This change did not improve execution time. Second, the DFA.nextWord structure was place in Shared Memory. The limited size of the Shared Memory limited the number of blocks that could concurrently execute on each SM. Finally, the DFA.nextWord structure was place in Constant Memory. Again, this scheme resulted in a slight degradation of the performance. Overall, this experiment demonstrated that for these sparsely accessed, read-only structures the Texture Memory is the optimal storage. The reason is that the texture memory is designed to support texture accesses in graphic, and these texture accesses have very similar access characteristics compared to the DFA accesses. Finally, the organization of the DFA structure could be revised to attempt to shrink the total storage to allow more efficient use of shared memory.

The data movement can likely be improved using the two-level results buffer used by the GPU-BLASTP implementation, and potential compression of the subject database could also help reduce the overhead of the copy.

The divergence analysis and potential improvement requires significant analysis and is beyond the scope of this report.

8.2 Future Directions

In addition to the ideas presented in this paper, future implementations of BLASTP on GPUs could examine the following ideas. First, multiple query sequences could be processed by the GPU for each block of the subject database sent to the GPU. This would allow the cost of copying the database to the GPU to be amortized across multiple searches. Second, the new CUDA Nested Parallelism feature could allow the `Blast_Kernel` to directly call the `sub_based_sw` kernel. This would also require the GPU to do the post processing of the `Blast_Kernel` results, but would eliminate the copies to and from the CPU between these two kernel stages of the algorithm. Finally, the results buffers could be stored in CPU memory. This would allow the results data to be written back to CPU memory as they are produced rather than at the end of the computation. Overall, GPUs appear to have significant potential for improving the performance of the BLASTP algorithm.

9.0 References

[1] T.F. Smith and M.S. Waterman, “Identification of Common Molecular Subseqeunces,” *J. Molecular Biology,* Vol 147, pp. 195-197, 1981.

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

[3] Xiao, S. and Lin, H. and Feng, W. “Accelerating Protein Sequence Search in a Heterogeneous Computing System”. *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International.*

[4] Cameron, M., Williams H.E., Cannane A. “A deterministic finite automaton for faster protein hit detection in BLAST”. Comput Biol. 2006 May, 13(4):965-78.

[5] NVIDIA GTX 680 Whitepaper. Retrieved June 5, 2012 from http://www.nvidia.es/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

[6] Panagiotis D. Vouzis *and* Nikolaos V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," Vol. 27, no. 2, pages 182-188, Bioinformatics, 2011 (Open Access).