GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU

Xinyan Zha and Sartaj Sahni, *Fellow*, *IEEE*

Abstract—We develop GPU adaptations of the Aho-Corasick and multipattern Boyer-Moore string matching algorithms for the two cases GPU-to-GPU (input to the algorithms is initially in GPU memory and the output is left in GPU memory) and host-to-host (input and output are in the memory of the host CPU). For the GPU-to-GPU case, we consider several refinements to a base GPU implementation and measure the performance gain from each refinement. For the host-to-host case, we analyze two strategies to communicate between the host and the GPU and show that one is optimal with respect to runtime while the other requires less device memory. This analysis is done for GPUs with one I/O channel to the host as well as those with 2. Experiments conducted on an NVIDIA Tesla GT200 GPU that has 240 cores running off of a Xeon 2.8 GHz quad-core host CPU show that, for the GPU-to-GPU case, our Aho-Corasick GPU adaptation achieves a speedup between 8.5 and 9.5 relative to a single-thread CPU implementation and between 2.4 and 3.2 relative to the best multithreaded implementation. For the host-to-host case, the GPU AC code achieves a speedup of 3.1 relative to a single-threaded CPU implementation. However, the GPU is unable to deliver any speedup relative to the best multithreaded code running on the quad-core host. In fact, the measured speedups for the latter case ranged between 0.74 and 0.83. Early versions of our multipattern Boyer-Moore adaptations ran 7 to 10 percent slower than corresponding versions of the AC adaptations and we did not refine the multipattern Boyer-Moore codes further.

Index Terms— Multipattern string matching, Aho-Corasick, multipattern Boyer-Moore, GPU, CUDA

1 Introduction

In multipattern string matching, we are to report all occurrences of a given set or dictionary of patterns in a target string. Multipattern string matching arises in a number of applications including network intrusion detection, digital forensics, business analytics, and natural language processing. For example, the popular open-source network intrusion detection system Snort [28] has a dictionary of several thousand patterns that are matched against the contents of Internet packets and the open-source file carver Scalpel [24] searches for all occurrences of headers and footers from a dictionary of about 40 header/footer pairs in disks that are many gigabytes in size. In both applications, the performance of the multipattern matching engine is paramount. In the case of Snort, it is necessary to search for thousands of patterns in relatively small packets at Internet speed while in the case of Scalpel we need to search for tens of patterns in hundreds of gigabytes of disk data.

Snort [28] employs the Aho-Corasick (AC) [1] multipattern search method while Scalpel [24] uses the Boyer-Moore single pattern search algorithm [4]. Since Scalpel uses a single pattern search algorithm, its runtime is linear in the product of the number of patterns in the pattern dictionary and the length of the target string in which the search is being done. Snort, on the other hand, because of its use of an efficient multipattern search algorithm has a runtime that is independent of the number of patterns in the dictionary and linear in the length of the target string.

Several researchers have attempted to improve the performance of multistring matching applications through the use of parallelism. For example, Scarpazza et al. [25], [26] port the deterministic finite automata version of the AC method to the IBM cell broadband engine (CBE) while Zha et al. [37] port a compressed form of the nondeterministic finite automata version of the AC method to the CBE. Jacob et al. [15] port Snort to a GPU. However, in their port, they replace the AC search method employed by Snort with the Knuth-Morris-Pratt [16] single-pattern matching algorithm. Specifically, they search for 16 different patterns in a packet in parallel employing 16 GPU cores. Huang et al. [14] do network intrusion detection on a GPU based on the multipattern search algorithm of Wu and Manber [35]. Smith et al. [27] and Vasiliadis et al. [31] use deterministic finite automata and extended deterministic finite automata to do regular expression matching on a GPU for intrusion detection applications. Marziale et al. [18] propose the use of GPUs and massive parallelism for in-place file carving. However, Zha and Sahni [38] show that the performance of an in-place file carver is limited by the time required to read data from the disk rather than the time required to search for headers and footers (when a fast multipattern matching algorithm is used). Hence, by doing asynchronous disk reads, the pattern matching time is effectively overlapped by the disk read time and the total time for the in-place carving operation equals that of the disk read time. Therefore, this application cannot benefit from the use of a GPU to accelerate pattern matching.

Our focus in this paper is accelerating the AC and Boyer-Moore multipattern string matching algorithms

• *The authors are with the Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611. E-mail: {xzha, sahni}@cise.ufl.edu.*

Manuscript received 21 Oct. 2011; accepted 16 Feb. 2012; published online 28 Feb. 2012.
Recommended for acceptance by K. Ghose.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-10-0752.
Digital Object Identifier no. 10.1109/TC.2012.61.

through the use of a GPU. A GPU operates in traditional master-slave fashion (see [23], for example) in which the GPU is a slave that is attached to a master or host processor under whose direction it operates. Algorithm development for master-slave systems is affected by the location of the input data and where the results are to be left. Generally, four cases-slave-to-slave, host-to-host, host-to-slave, and slave-to-host-arise depending on whether the input (output) reside initially (at the end) in the slave (GPU) or the master memory (CPU) [32], [33], [34]. In this paper, we address the first two cases only. In our context, we refer to the first case (slave-to-slave) as GPU-to-GPU.

GPU implementations of the AC algorithm have been proposed earlier [17], [30], [39] and a GPU implementation of the Boyer-Moore multistring matching algorithm is described in [39]. Lin et al. [17] and Tumeo et al. [30] consider the host-to-host case in which the target string begins in the host CPU and the pattern matches are to be brought back from the GPU to the host CPU. The AC GPU implementation of Lin et al. [17] uses one thread for each position in the target string. This thread determines whether its assigned position is the start of match. This implementation, which is called PFAC (Parallel Failureless-AC), achieved a throughput of almost 4 GBps (Giga bytes per second) on a GTX295 GPU that has 480 CUDA cores. The PFAC code is available in the PFAC library [21]. In the GPU implementations of Tumeo et al. [30] and Zha and Sahni [39] each thread is assigned a portion of the target string to search. There is sufficient overlap among the portions assigned to different threads so that matches that cross portion boundaries are not missed. Although this approach was described in [17], it was rejected because of the overhead associated with portion overlaps. Tumeo et al. [30] experimented with the NVIDIA Tesla C1060 and the Fermi C2050 GPUs. On dictionaries with at least 20K patterns, they achieve a throughput between 1.5 and 12 GBps using a single C1060 and a throughput between 3.3 and 22.7 GBps using a single C2050. The performance varies with the pattern set as well as with the number of pattern matches found. Further, four C1060s were able to match the performance of a single C2050 while four C2050s were able to achieve a throughput between 2 and 3 times that of a single C2050. Zha and Sahni [39], which is a preliminary version of this current paper, considers the GPU-to-GPU case and achieves a throughput of 30 GBps on a GT200 using a dictionary with 33 patterns. Their Boyer-Moore implementation achieves a throughput of 6 GBps.

The contributions of this paper are:

1. Careful analysis of the deficiencies of a base implementation of the GPU-to-GPU AC and Boyer-Moore algorithms and a clear description of how these deficiencies may be overcome. Overcoming the noted deficiencies speeds the AC implementation by a factor between 8 and 9. For the Boyer-Moore algorithm, we did only the first step of a 4-step deficiency elimination process and achieved a speedup of almost 2. The remaining steps were not done because it became evident that the remaining steps would not make the GPU-to-GPU Boyer-Moore algorithm competitive with the optimized AC algorithm.
2. For host-to-host computing, we consider two strategies to overlap input-output data transfer between host and GPU with GPU computation. The first of these, which is intuitive and given in [10], requires more GPU memory than is required by the other. The performance of each strategy is analyzed for master-slave systems that have either 1 (e.g., GT200 and C1060) or 2 (e.g., C2050) I/O channels between master and slave. We prove the optimality of the first strategy for systems with 1 and 2 I/O channels and the suboptimality of the second strategy. For the second strategy, we establish tight performance bounds relative to optimal performance. These bounds enable the programmer to evaluate the tradeoff between GPU memory requirement and overall application runtime and to possibly use a hybrid strategy to optimize runtime subject to the constraint of available GPU memory.
3. We couple our first data transfer strategy with our optimized GPU-to-GPU AC implementation to arrive at a host-to-host implementation that achieves a throughput of 10 GBps on a GT200. This throughput is at the high end of the 1.5 to 12 GBps performance achieved by the C1060, which is in the same family as the GT200, implementation of [30].

The remainder of this paper is organized as follows: In Section 2, we provide a brief review of single and multipattern string matching with a focus on the AC and Boyer-Moore algorithms. Sections 3 and 4 describe our GPU adaptation of these matching algorithms for the GPU-to-GPU and host-to-host cases. Section 5 discusses our experimental results and we conclude in Section 6. The reader unfamiliar with NVIDIA's Tesla architecture and CUDA is referred to [10] for details.

2 Boyer-Moore and AC String Matching

Knuth et al. [16] developed the first linear time algorithm for string matching and Aho and Corasick [1] did this for multistring matching. Yao [36] has shown that the average complexity of the string matching problem is $\Omega(|S|\log_c(|P|)/|P|)$, where P is the pattern, S is the target string, and c is the size of the alphabet. Navarro and Fredericksson [19] have generalized this bound to $\Omega(|S|\log_c(rm)/m)$, where m is the length of the shortest pattern, for multipattern string matching. Both lower bounds are tight [7], [8], [9].

Boyer-Moore [4], Horspool [13], and Galil [12] have developed pattern matching algorithms Central to these algorithms is a *bad character function* for P that specifies how many characters to shift P right before reexamining pairs of characters from P and S for a match. More specifically, the bad character function for P gives the distance from the end of P of the last occurrence of each possible character that may appear in S. In practice, many of the shifts in the bad character function of a pattern are close to the length, $|P|$, of the pattern P making the algorithms of [4], [13], [12] very fast in practice. In fact, when the alphabet size is large compared to $|P|$, the average runtime of the Boyer-Moore algorithm is $O(|S|/|P|)$. Galil's [12] variation has a worst case runtime that is $O(|S|)$ and Horspool's [13] simplification of the Boyer-Moore algorithm has a performance that is about the same as that of the Boyer-Moore algorithm.

Several multipattern extensions to the Boyer-Moore search algorithm have been proposed [2], [6], [11], [35]. All of these multipattern search algorithms extend the bad character function for a single pattern to a bad character function for a set of patterns. The Set-wise Boyer-Moore algorithm of [11] performs multipattern matching using this combined bad function. The multipattern search algorithms of [2], [6], [35] employ additional techniques to speed the search further. The average runtime of the algorithms of [2], [6], [35] is $O(|S|/minL)$, where $minL$ is the length of the shortest pattern.

The multipattern Boyer-Moore algorithm used by us is due to [6]. This algorithm employs two additional functions $shift1$ and $shift2$ and a trie, called the reverse trie, which contains the reverse of the patterns.

There are two versions-nondeterministic and deterministic -of the AC [1] multipattern matching algorithm. Both versions use a finite state machine/automaton to represent the dictionary of patterns.[1] In the deterministic version (DFA), which is the version we use in this paper, each state of the finite automaton has a well-defined transition for every character in the alphabet as well as a list of matched patterns. The search starts with the automaton start state designated as the current state and the first character in the text string, S, that is being searched designated as the current character. At each step, a state transition is made by examining the current character of S. A transition to the state corresponding to the current character is made and the next character of S becomes the current character. Whenever a state transition is made, the patterns in the list of matched patterns for the reached state are output along with the position in S of the current character. This output is sufficient to identify all occurrences, in S, of all dictionary patterns. Aho and Corasick [1] show how to compute the DFA for a set of patterns. The number of state transitions made by the DFA when searching for matches in a string of length n is n. In the nondeterministic (NFA) version finite automata states have two kinds of transitions success and failure. Success transitions are defined for characters that match a pattern character; for the remaining characters, a failure transition is made. When the NFA version is used, the number of state transitions in $2n$. The NFA version, however, uses less memory as finite automata states have few success transitions and so may be better compacted than DFA states. However, the NFA version results in thread divergence when failure transitions occur.

3 GPU-TO-GPU

3.1 Strategy

The input to the multipattern matcher is a character array $input$ and the output is an array $output$ of states or reverse-trie node indexes (or pointers). Both arrays reside in device memory. When the AC algorithm is used, $output[i]$ gives the state of the AC DFA following the processing of $input[i]$. Since every state of the AC DFA contains a list of patterns that are matched when this state is reached, $output[i]$ enables us to determine all matching patterns that end at input character i. When the multipattern Boyer-Moore (mBM) algorithm is used, $output[i]$ is the last reverse trie node visited over all examinations of $input[i]$. Using this information and the pattern list stored in the trie node, we may determine all pattern matches that begin at $input[i]$. If we assume that the number of states in the AC DFA as well as the number of nodes in the mBM reverse trie is no more than 65,536, a state/node index can be encoded using 2 bytes and the size of the output array is twice that of the input array.

n	number of characters in string to be searched
$maxL$	length of longest pattern
S_{block}	number of input characters for which a thread block computes output
B	number of blocks = $n/Sblock$
T	number of threads in a thread block
S_{thread}	number of input characters for which a thread computes output = S_{block}/T
$tWord$	$S_{thread}/4$
TW	total work = effective string length processed

Fig. 1. GPU-to-GPU notation.

Our computational strategy is to partition the output array into blocks of size S_{block} (Fig. 1 summarizes the notation used in this section). The blocks are numbered (indexed) 0 through n/S_{block}, where n is the number of output values to be computed. Note that n equals the number of input characters as well. $output[i{*}S_{block} : (i+1){*}S_{block}-1]$ comprises the ith output block. To compute the ith output block, it is sufficient for us to use AC on $input[b{*}S_{block} - maxL + 1 : (b+1){*}S_{block} - 1]$, where $maxL$ is the length of the longest pattern (for simplicity, we assume that there is a character that is not the first character of any pattern and set $input[-maxL+1:-1]$ equal to this character) or mBM on $input[b{*}S_{block} : (b+1){*}S_{block} + maxL - 2]$ (we may assume that $input[n : n + maxL - 2]$ equals a character that is not the last character of any pattern). So, a block actually processes a string whose length is $S_{block} + maxL - 1$ and produces S_{block} elements of the output. The number of blocks is $B = n/S_{block}$.

Suppose that an output block is computed using T threads. Then, each thread could compute $S_{thread} = S_{block}/T$ of the output values to be computed by the block. So, thread t (thread indexes begin at 0) of block b could compute

$$output[b{*}S_{block} + t{*}S_{thread} : b{*}S_{block} + t{*}S_{thread} + S_{thread} - 1]$$

For this, thread t of block b would need to process the substring

$$input[b{*}S_{block} + t{*}S_{thread} - maxL + 1 : b{*}S_{block} + t{*}S_{thread} + S_{thread} - 1]$$

when AC is used and $input[b{*}S_{block} + t{*}S_{thread} : b{*}S_{block} + t{*}S_{thread} + S_{thread} + maxL - 2]$ when mBM is used. Fig. 2 gives the pseudocode for a T-thread computation of block i of the output using the AC DFA. The variables used are self-explanatory and the correctness of the pseudocode follows from the preceding discussion.

As discussed earlier, the arrays $input$ and $output$ reside in device memory. The AC DFA (or the mBM reverse tries

1. The use of the terms deterministic and nondeterministic in the intrusion detection literature for the two versions of the AC algorithm is unfortunate as both versions of the algorithm actually use deterministic finite state machines/automata.

Algorithm *basic*
// compute block b of the output array using T threads and AC
// following is the code for a single thread, thread t, $0 \leq t < T$
t = thread index;
b = block index;
$state$ = 0; // initial DFA state
$outputStartIndex = b * S_{block} + t * S_{thread}$;
$inputStartIndex = outputStartIndex - maxL + 1$;

// process $input[inputStartIndex : outputStartIndex - 1]$
for (**int** $i = inputStartIndex$; $i < outputStartIndex$; $i++$)
  $state = nextState(state, input[i])$;

//compute output
for (**int** $i = outputStartIndex$; $i < outputStartIndex + S_{thread}$; $i++$)
  $output[i] = state = nextState(state, input[i])$;
end;

Fig. 2. Overall GPU-to-GPU strategy using AC.

in case the mBM algorithm is used) resides in texture memory because texture memory is cached and is sufficiently large to accommodate the DFA (reverse trie). While shared and constant memories will result in better performance, neither is large enough to accommodate the DFA (reverse trie). Note that each state of a DFA has A transitions, where A is the alphabet size. For ASCII, $A = 256$. Assuming that the total number of states is fewer than 65,536, each state transition of a DFA takes 2 bytes. So, a DFA with d states requires $512d$ bytes. In the 16 KB shared memory that our Tesla has, we can store at best a 32-state DFA. The constant memory on the Tesla is 64 KB. So, this can handle, at best, a 128-state DFA. Since the nodes of the mBM reverse trie are as large as a DFA state, it is not possible to store the reverse trie for any reasonable pattern dictionary in shared or constant memory either. Each of the mBM shift functions, $shift1$ and $shift2$, need 2 bytes per reverse-trie node. So, our shared memory can store these functions when the number of nodes does not exceed 4K; constant memory may be used for tries with fewer than 16K nodes. The bad character function $B()$ has 256 entries when the alphabet size is 256. This function may be stored in shared memory.

A nice feature of Algorithm *basic* is that all T threads that work on a single block can execute in lock-step fashion as there is no divergence in the execution paths of these T threads. This makes it possible for an SM of a GPU to efficiently compute an output block using T threads. With 30 SMs, we can compute 30 output blocks at a time. The pseudocode of Fig. 2 does, however, have deficiencies that are expected to result in nonoptimal performance on a GPU. These deficiencies are described below.

Deficiency D1. Since the input array resides in device memory, every reference to the array $input$ requires a device memory transaction (in this case a read). There are two sources of inefficiency when the read accesses to $input$ are actually made on the Tesla GPU—1) Our Tesla GPU performs device-memory transactions for a half-warp (16) of threads at a time. The available bandwidth for a single transaction is 128 bytes. Each thread of our code reads 1 byte. So, a half-warp reads 16 bytes. Hence, barring any other limitation of our GPU, our code will utilize 1/8th the available bandwidth between device memory and an SM. 2) The Tesla is able to coalesce the device memory transactions from several threads of a half-warp into a single transaction. However, coalescing occurs only when the device-memory accesses of two or more threads in a half-warp lie in the same 128-byte segment of device memory. When $S_{thread} > 128$, the values of $inputStartIndex$ for consecutive threads in a half-warp (note that two threads $t1$ and $t2$ are in the same half-warp iff $\lfloor t1/16 \rfloor = \lfloor t2/16 \rfloor$) are more than 128 bytes apart. Consequently, for any given value of the loop index i, the read accesses made to the array $input$ by the threads of a half-warp lie in different 128-byte segments and so no coalescing occurs. Although the pseudocode is written to enable all threads to simultaneously access the needed input character from device memory, an actual implementation on the Tesla GPU will serialize these accesses and, in fact, every read from device memory will transmit exactly 1 byte to an SM resulting in a 1/128 utilization of the available bandwidth.

Deficiency D2. The writes to the array $output$ suffer from deficiencies similar to those identified for the reads from the array $input$. Assuming that our DFA has no more than $2^{16} = 65{,}536$ states, each state can be encoded using 2 bytes. So, a half-warp writes 64 bytes when the available bandwidth for a half-warp is 128 bytes. Further, no coalescing takes place as no two threads of a half-warp write to the same 128-byte segment. Hence, the writes get serialized and the utilized bandwidth is 2 bytes, which is 1/64th of the available bandwidth.

3.1.1 Analysis of Total Work

Using the GPU-to-GPU strategy of Fig. 2, we essentially do multipattern searches on $B*T$ strings of length $S_{thread} + maxL - 1$ each. With a linear complexity for multipattern search, the total work, TW, is roughly equivalent to that done by a sequential algorithm working on an input string of length

$$\begin{aligned} TW &= B*T*(S_{thread} + maxL - 1) \\ &= n*\left(1 + \frac{1}{S_{thread}} * (maxL - 1)\right). \end{aligned}$$

So, our GPU-to-GPU strategy incurs an overhead of $\frac{1}{S_{thread}} * (maxL - 1) * 100\%$ in terms of the effective length of the string that is to be searched. Clearly, this overhead varies substantially with the parameters $maxL$ and S_{thread}. Suppose that $maxL = 17$, $S_{block} = 14{,}592$, and $T = 64$ (as in our experiments of section 5). Then, $S_{thread} = 228$ and $TW = 1.07n$. The overhead is 7 percent.

3.2 Addressing the Deficiencies

3.2.1 Deficiency D1-Reading from Device Memory

A simple way to improve the utilization of available bandwidth between the device memory and an SM is to have each thread input 16 characters at a time, process these 16 characters, and write the output values for these 16 characters to device memory. For this, we will need to

```
// define space in shared memory to store the input data
_shared_ unsigned char sInput[S_block + maxL - 1];

// typecast to uint4
uint4 *sInputUint4 = ( uint4 *)sInput;

// read as uint4s, assume S_block and maxL - 1 are divisible by 16
int numToRead = (S_block + maxL - 1)/16;
int next = b * S_block/16 - (maxL - 1)/16 + t;

// T threads collectively input a block
for (int i = t; i < numToRead; i+ = T, next+ = T)
    sInputUint4[i] = inputUint4[next];
```

Fig. 3. T threads collectively read a block and save in shared memory.

cast the input array from its native data type `unsigned char` to the data type `uint4` as below:

```
uint4 * inputUint4 = (uint4 *) input;
```

A variable `var` of type `uint4` is comprised of four unsigned 4-byte integers `var.x`, `var.y`, `var.z`, and `var.w`. The statement

```
uint4 in4 = inputUint4[i];
```

reads the 16 bytes `input[16*i:16*i+15]` and stores these in the variable `in4`, which is assigned space in shared memory. Since the Tesla is able to read up to 128 bits (16 bytes) at a time for each thread, this simple change increases bandwidth utilization for the reading of the input data from 1/128 of capacity to 1/8 of capacity! However, this increase in bandwidth utilization comes with some cost. To extract the characters from `in4` so they may be processed one at a time by our algorithm, we need to do a shift and mask operation on the four components of `in4`. We shall see later that this cost may be avoided by doing a recast to `unsigned char`.

Since a Tesla thread cannot read more than 128 bits at a time, the only way to improve bandwidth utilization further is to coalesce the accesses of multiple threads in a half-warp. To get full bandwidth utilization at least eight threads in a half-warp will need to read `uint4`s that lie in the same 128-byte segment. However, the data to be processed by different threads do not lie in the same segment. To get around this problem, threads cooperatively read all the data needed to process a block, store this data in shared memory, and finally read and process the data from shared memory. In the pseudocode of Fig. 3, T threads cooperatively read the input data for block b. This pseudocode, which is for thread t operating on block b, assumes that S_{block} and $maxL-1$ are divisible by 16 so that a whole number of `uint4`s are to be read and each read begins at the start of a `uint4` boundary (assuming that $input[-maxL+1]$ begins at a `uint4` boundary). In each iteration (except possibly the last one), T threads read a consecutive set of T `uint4`s from device memory to shared memory and each `uint4` is 16 input characters.

In each iteration (except possibly the last one) of the for loop, a half-warp reads 16 adjacent `uint4`s for a total of 256 adjacent bytes. If $input[-maxL+1]$ is at a 128-byte boundary of device memory, S_{block} is a multiple of 128, and T is a multiple of 8, then these 256 bytes fall in 2 128-byte segments and can be read with two memory transactions. So, bandwidth utilization is 100 percent. Although 100 percent utilization is also obtained using `uint2`s (now each thread reads 8 bytes at a time rather than 16 and a half-warp reads 128 bytes in a single memory transaction), the observed performance is slightly better when a half-warp reads 256 bytes in two memory transactions.

Once we have read the data needed to process a block into shared memory, each thread may generate its share of the output array as in Algorithm $basic$ but with the reads being done from shared memory. Thread t will need

$$sInput[t * S_{thread} : (t+1) * S_{thread} + maxL - 2]$$

or $sInputUint4[t * S_{thread}/16{:}(t+1) * S_{thread}/16 + \lceil (maxL - 1)/16 \rceil - 1]$, depending on whether a thread reads the input data from shared memory as characters or as `uint4`s. When the latter is done, we need to do shifts and masks to extract the characters from the four unsigned integer components of a `uint4`.

Although the input scheme of Fig. 3 succeeds in reading in the data utilizing 100 percent of the bandwidth between device memory and an SM, there is potential for shared-memory bank conflicts when the threads read the data from shared memory. Shared memory is partitioned into 16 banks. The ith 32-bit word of shared memory is in bank $i \bmod 16$. For maximum performance, the threads of a half-warp should access data from different banks. Suppose that $S_{thread} = 224$ and $sInput$ begins at a 32-bit word boundary. Let $tWord = S_{thread}/4$ ($tWord = 224/4 = 56$ for our example) denote the number of 32-bit words processed by a thread exclusive of the additional $maxL - 1$ characters needed to properly handle the boundary. In the first iteration of the data processing loop, thread t needs $sInput[t * S_{thread}]$, $0 \le t < T$. So, the words accessed by the threads in the half-warp $0 \le t < 16$ are $t * tWord$, $0 \le t < 16$ and these fall into banks $(t * tWord) \bmod 16$, $0 \le t < 16$. For our example, $tWord = 56$ and $(t * 56) \bmod 16 = 0$ when t is even and $(t * 56) \bmod 16 = 8$ when t is odd. Since each bank is accessed eight times by the half-warp, the reads by a half-warp are serialized to eight shared memory accesses. Further, since on each iteration, each thread steps right by one character, the bank conflicts remain on every iteration of the process loop. We observe that whenever $tWord$ is even, at least threads 0 and 8 access the same bank (bank 0) on each iteration of the process loop. Theorem 1 shows that when $tWord$ is odd, there are no shared-memory bank conflicts.

Theorem 1. *When $tWord$ is odd, $(i * tWord) \bmod 16 \neq (jk) \bmod 16$, $0 \le i < j < 16$.*

Proof. The proof is by contradiction. Assume there exist i and j such that $0 \le i < j < 16$ and $(i * tWord) \bmod 16 = (j * tWord) \bmod 16$. For this to be true, there must exist nonnegative integers a, b, and c, $a < c$, $0 \le b < 16$ such that $i * tWord = 16a + b$ and $j * tWord = 16c + b$. So, $(j - i) * tWord = 16(c - a)$. Since $tWord$ is odd and $c - a > 0$, $j - i$ must be divisible by 16. However, $j - i < 16$ and so cannot be divisible by 16. This contradiction implies that our assumption is invalid and the theorem is proved. □

It should be noted that even when $tWord$ is odd, the input for every block begins at a 128-byte segment of device memory (assuming that for the first block begins at a 128-byte segment) provided T is a multiple of 32. To see this, observe that $S_{block} = 4 * T * tWord$, which is a multiple of 128 whenever T is a multiple of 32. As noted earlier, since the Tesla schedules threads in warps of size 32, we normally would choose T to be a multiple of 32.

3.2.2 Deficiency D2-Writing to Device Memory

We could use the same strategy used to overcome deficiency D1 to improve bandwidth utilization when writing the results to device memory. This would require us to first have each thread write the results it computes to shared memory and then have all threads collectively write the computed results from shared memory to device memory using `uint4`s. Since the results take twice the space taken by the input, such a strategy would necessitate a reduction in S_{block} by two-thirds. For example, when $maxL = 17$, and $S_{block} = 14,6592$ we need 14,608 bytes of shared memory for the array $sInput$. This leaves us with a small amount of 16 KB shared memory to store any other data that we may need to. If we wish to store the results in shared memory as well, we must use a smaller value for S_{block}. So, we must reduce S_{block} to about 14,592/3 or 4,864 to keep the amount of shared memory used the same. When $T = 64$, this reduction in block size increases the total work overhead from approximately 7 percent to approximately 22 percent. We can avoid this increase in total work overhead by first having each thread processes the first $maxL - 1$ characters it is to process. This generates no output and so we need no memory to store output.

Next, each thread reads the remaining S_{thread} characters of input data it needs from shared memory to registers. For this, we declare a register array of unsigned integers and typecast $sInput$ to unsigned integer. Since, the T threads have a total of 16,384 registers, we have sufficient registers provided $S_{block} \leq 4 * 16384 = 64$K (in reality, S_{block} would need to be slightly smaller than 64K as registers are needed to store other values such as loop variables). Since total register memory exceeds the size of shared memory, we always have enough register space to save the input data that is in shared memory.

Unless $S_{block} \leq 4{,}864$, we cannot store all the results in shared memory. However, to do 128-byte write transactions to device memory, we need only sets of 64 adjacent results (recall that each result is 2 bytes). So, the shared memory needed to store the results is $128T$ bytes. Since we are contemplating $T = 64$, we need only 8K of shared memory to store the results from the processing of 64 characters per thread. Once each thread has processed 64 characters and stored these in shared memory, we may write the results to device memory. The total number of outputs generated by a thread is $S_{thread} = 4 * tWord$. These outputs take a total of $8 * tWord$ bytes. So, when $tWord$ is odd (as required by Theorem 1), the output generated by a thread is a nonintegral number of `uint4`s (recall that each `uint4` is 16 bytes). Hence, the output for some of the threads does not begin at the start of a `uint4` boundary of the device array $output$ and we cannot write the results to device memory as `uint4`s. Rather, we need to write as `uint2`s (a thread generates an integral number $tWord$ of `uint2`s). With each thread writing a `uint2`, it takes 16 threads to write 128 bytes of output from that thread. So, T threads can write the output generated from the processing of 64 characters/thread in 16 rounds of `uint2` writes.

One difficulty is that, as noted earlier, when $tWord$ is odd, even though the segment of device memory to which the output from a thread is to be written begins at a `uint2` boundary, it does not begin at a `uint4` boundary. This means also that this segment does not begin at a 128-byte boundary (note that every 128-byte boundary is also a `uint4` boundary). So, even though a half-warp of 16 threads is writing to 128 bytes of contiguous device memory, these 128-bytes may not fall within a single 128-byte segment. When this happens, the write is done as two memory transactions.

The described procedure to handle 64 characters of input per thread is repeated $\lceil S_{thread}/64 \rceil$ times to complete the processing of the entire input block. In case S_{thread} is not divisible by 64, each thread produces fewer than 64 results in the last round. For example, when $S_{thread} = 228$, we have a total of four rounds. In each of the first three rounds, each thread processes 64 input characters and produces 64 results. In the last round, each thread processes 36 characters and produces 36 results. In the last round, groups of threads either write to contiguous device memory segments of size 64 or 8 bytes and some of these segments may span 2 128-byte segments of device memory.

As we can see, using an odd $tWord$ is required to avoid shared-memory bank conflicts but using an odd $tWord$ (actually using a $tWord$ value that is not a multiple of 16) results in suboptimal writes of the results to device memory. To optimize writes to device memory, we need to use a $tWord$ value that is a multiple of 16. Since the Tesla executes threads on an SM in warps of size 32, T would normally be a multiple of 32. Further, to hide memory latency, it is recommended that T be at least 64. With $T = 64$ and a 16 KB shared memory, S_{thread} can be at most $16 * 1,024/64 = 256$ and so $tWord$ can be at most 64. However, since a small amount of shared memory is needed for other purposes, $tWord < 64$. The largest value possible for $tWord$, that is, a multiple of 16 is therefore 48. The total work, TW, when $tWord = 48$ and $maxL = 17$ is $n * (1 + \frac{1}{4*48} * 16) = 0.083n$. Compared to the case $tWord = 57$, the total work overhead increases from 7 to 8.3 percent. Whether we are better off using $tWord = 48$, which results in optimized writes to device memory but shared-memory bank conflicts and larger work overhead, or with $tWord = 57$, which has no shared-memory bank conflicts and lower work overhead but suboptimal writes to device memory, can be determined experimentally.

4 Host-to-Host

4.1 Strategies

Since the GPUs support asynchronous transfer of data between device memory and pinned host memory, it is possible to overlap the time spent in data transfer to and from the device with the time spent by the GPU in computing the results. There are at least two ways to accomplish the overlap of I/O between host and device and GPU computation. In Strategy A (Fig. 4), which is given in [10], we have three loops. The first loop asynchronously writes the input data to device memory

for (**int** $i = 0;\ i < numOfSegments;\ i++$)
Asynchronously write segment i from host to device using stream i;

for (**int** $i = 0;\ i < numOfSegments;\ i++$)
Process segment i on the GPU using stream i;

for (**int** $i = 0;\ i < numOfSegments;\ i++$)
Asynchronously read segment i results from device using stream i;

Fig. 4. Host-to-host strategy A.

in segments, the second processes each segment on the GPU, and third reads the results for each segment back from device memory asynchronously. To ensure that the processing of a segment does not begin before the asynchronous transfer of that segments data from host to device completes and also that the reading of the results for a segment begins only after the completion of the processing of the segment, CUDA provides the concept of a stream. Within a stream, tasks are done in sequence. With reference to Fig. 4, the number of streams equals the number of segments and the tasks in the ith stream are: write segment i to device memory, process segment i, read the results for segment i from device memory. To get the correct results, each segment sent to the device memory must include the additional $maxL - 1$ characters needed to detect matches that cross segment boundaries.

For strategy A to work, we must have sufficient device memory to accommodate the input data for all segments as well as the results from all segments. Fig. 5 gives an alternative strategy that requires only sufficient device memory for two segments (two input buffers IN0 and IN1 and two output buffers OUT0 and OUT1). In this strategy, the GPU processes input data that is in IN0 (IN1) and writes the results to OUT0 (OUT1). While the GPU is using buffers IN0 and OUT0 (or IN1 and OUT1) in this way, the host writes to IN1 and reads from OUT1 (or IN0 and OUT0). We could, of course, couple strategies A and B to obtain a hybrid strategy.

Write segment 0 from host to device buffer IN0;
for (**int** $i = 1;\ i < numOfSegments;\ i++$)
{
Asynchronously write segment i from host to device buffer IN1;
Process segment $i - 1$ on the GPU using IN0 and OUT0;
Wait for all read/write/compute to complete;
Asynchronously read segment $i - 1$ results from OUT0;
Swap roles of IN0 and IN1;
Swap roles of OUT0 and OUT1;
}
Process the last segment on the GPU using IN0 and OUT0;
Read last segment's results from OUT0;

Fig. 5. Host-to-host strategy B.

s	number of segments
t_w	time to write an input data segment from host to device memory
t_r	time to read an output data segment from device to host memory
t_p	time taken by GPU to process an input data segment, create corresponding output segment
T_w	$\sum_{i=0}^{s-1} t_w = s * t_w$
T_r	$\sum_{i=0}^{s-1} t_r = s * t_r$
T_p	$\sum_{i=0}^{s-1} t_p = s * t_p$
$T_s^w(i)$	time at which the writing of input segment i to device memory starts
$T_s^p(i)$	time at which the processing of segment i by the GPU starts
$T_s^r(i)$	time at which the reading of output segment i to host memory starts
$T_f^w(i)$	time at which the writing of input segment i to device memory finishes
$T_f^p(i)$	time at which the processing of segment i by the GPU finishes
$T_f^r(i)$	time at which the reading of output segment i to host memory finishes
T_A	completion time using strategy A
T_B	completion time using strategy B
L	lower bound on completion time

Fig. 6. Notation used in completion time analysis.

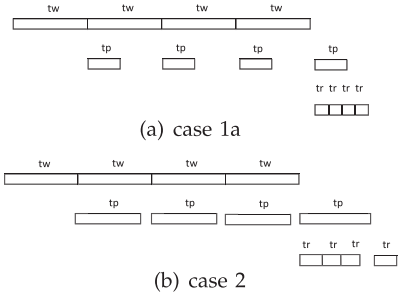
We analyze the relative time performance of these two host-to-host strategies in the next section.

4.2 Completion Time-One I/O Channel

In this section, we analyze the performance of strategies A and B for GPUs such as the GT200 and C1060 that have a single I/O channel to the host. In this case, it is not possible to overlap the transfer of input data from host to GPU with the transfer of results from GPU to host. Fig. 6 summarizes the notation used in our analysis of the completion time of strategies A and B.

For our analysis, we make several simplifying assumptions as below:

1. The time, t_w, to write or copy a segment of input data from the host to the device memory is the same for all segments.
2. The time, t_p, the GPU takes to process a segment of input data and create its corresponding output segment is the same for all segments.
3. The time, t_r, to read or copy a segment of output data from the host to the device memory is the same for all segments.
4. The write, processing, and read for each segment begins at the earliest possible time for the chosen strategy and completes t_w, t_p, and t_r units later, respectively.
5. In every feasible strategy, the relative order of segment writes, processing, and reads is the same and is segment 0, followed by segment $1, \ldots$, and ending with segment $s - 1$, where s is the number of segments.



(a) case 1a

(b) case 2

Fig. 7. Strategy A, $t_w \geq t_p$, $s = 4$ (cases 1a and 2).

Writing from the host memory to the device memory uses the same I/O channel/bus as used to read from the device memory to the host memory and the GPU is necessarily idle when the first input segment is being written to the device memory and the last output segment is being read from this memory. So, $t_w + \max\{(s-1)(t_w + t_r), s * t_p\} + t_r$ is a lower bound on the completion time of any host-to-host computing strategy.

It is easy to see that when the number of segments s is 1, the completion time for both strategies A and B is $t_w + t_p + t_r$, which equals the lower bound. Actually, when $s = 1$, both strategies are identical and optimal. The analysis of the two strategies for $s > 1$ is more complex and is done below in Theorems 2 to 5. We note that assumption 4 implies that $T_f^w(i) = T_s^w(i) + t_w$, $T_f^p(i) = T_s^p(i) + t_p$, and $T_f^r(i) = T_s^r(i) + t_r$, $0 \leq i < s$. The completion time is $T_f^r(s-1)$.

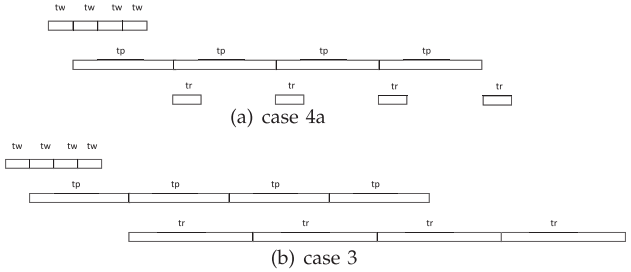
Theorem 2. *When $s > 1$, the completion time, T_A, of strategy A is:*

1. $T_w + T_r$ *whenever any of following holds:*
   a. $t_w \geq t_p \wedge t_p \leq T_r - t_r$
   b. $t_w < t_p \wedge T_w - t_w > t_p \wedge t_r \geq t_p$
   c. $t_w < t_p \wedge T_w - t_w > t_p \wedge t_r < t_p \wedge \nexists i, 0 \leq i < s[t_w + (i+1)t_p > T_w + it_r]$
2. $T_w + t_p + t_r$ *when* $t_w \geq t_p \wedge t_p > T_r - t_r$
3. $t_w + t_p + T_r$ *when* $t_w < t_p \wedge T_w - t_w \leq t_p \wedge t_r > t_p$
4. $t_w + T_p + t_r$ *when either of the following holds:*
   a. $t_w < t_p \wedge T_w - t_w \leq t_p \wedge t_r \leq t_p$
   b. $t_w < t_p \wedge T_w - t_w > t_p \wedge t_r < t_p \wedge \exists i, 0 \leq i < s[t_w + (i+1)t_p > T_w + i*t_r]$.

Proof. It should be easy to see that the conditions listed in the theorem exhaust all possibilities. When strategy A is used, all the writes to device memory complete before the first read begins (i.e., $T_s^r(0) \geq T_f^w(s-1)$), $T_s^w(i) = i*t_w, T_f^w(i) = (i+1)t_w, 0 \leq i < s$, and $T_s^r(0) \geq T_f^w(s-1) = s*t_w = T_w$.

When $t_w \geq t_p$, $T_s^p(i) = T_f^w(i) = (i+1)t_w$ (Fig. 7 illustrates this for $s = 4$). Hence,

$$T_f^p(i) = (i+1)t_w + t_p \leq T_f^w(i+1), 0 \leq i < s,$$



(a) case 4a

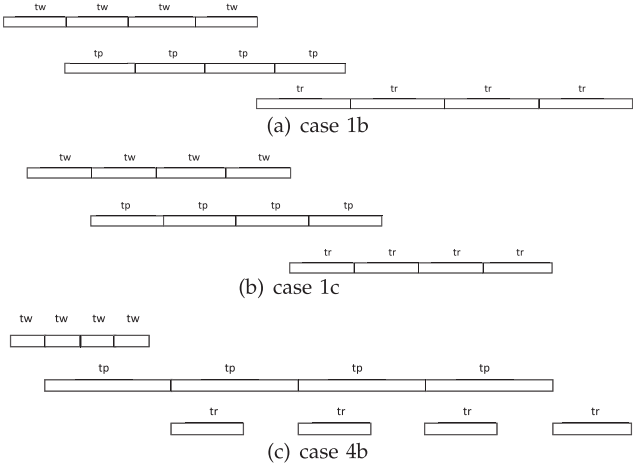
(b) case 3

Fig. 8. Strategy A, $t_w < t_p$, $T_w - t_w \leq t_p$, $s = 4$ (cases 4a and 3).

where $T_f^w(s)$ is defined to be $(s+1)t_w$. So, $T_s^r(0) = \max\{T_w, T_f^p(0)\} = T_w$ and $T_s^r(i) = \max\{T_f^r(i-1), T_f^p(i)\}$, $1 \leq i < s$. Since $T_f^p(i) \leq T_w, i < s-1$, $T_f^r(i) \geq T_f^r(i-1)$, $1 \leq i < s$, and $T_f^r(0) \geq T_w$, $T_s^r(i) = T_f^r(i-1)$, $1 \leq i < s-1$. So, $T_f^r(s-2) = T_s^r(s-2) + t_r = T_w + (s-1)t_r = T_w + T_r - t_r$. Hence, $T_s^r(s-1) = \max\{T_w + T_r - t_r, T_w + t_p\}$ and $T_A = T_f^r(s-1) = T_s^r(s-1) + t_r = \max\{T_w + T_r, T_w + t_p + t_r\}$. So, when $t_p \leq T_r - t_r$, $T_A = T_W + T_r$ (Fig. 7a) and when $t_p > T_r - t_r$, $T_A = T_w + t_p + t_r$ (Fig. 7b). This proves cases 1a and 2 of the theorem.

When $t_w < t_p$, $T_f^p(i) = t_w + (i+1)t_p$, $0 \leq i < s$. We consider the two subcases $T_w - t_w \leq t_p$ and $T_w - t_w > t_p$. The first of these has two subsubcases of its own-$t_r \leq t_p$ (theorem case 4a) and $t_r > t_p$ (theorem case 3). These subsubcases are shown in Fig. 8 for the case of four segments. It is easy to see that $T_A = t_w + T_p + t_r$ when $t_r \leq t_p$ and $T_A = t_w + t_p + T_r$ when $t_r > t_p$. The second subcase ($t_w < t_p$ and $T_w - t_w > t_p$) has two subsubcases as well-$t_r \geq t_p$ and $t_r < t_p$. When $t_r \geq t_p$ (Fig. 9a), $T_s^r(i) = T_w + i*t_r$, $0 \leq i < s$ and $T_A = T_f^r(s-1) = T_w + T_r$ (theorem case 1b). When $t_r < t_p \wedge \nexists i, 0 \leq i < s[t_w + (i+1)t_p > T_w + it_r]$ (theorem case 1c), $\nexists i$, $0 \leq i < s[T_f^p(i) > T_f^r(i-1)]$, where $T_f^r(-1)$ is defined to be T_w. So, $T_A = T_f^r(s-1) + t_r = T_w + T_r$ (Fig. 9b). When $t_r < t_p \wedge \exists i, 0 \leq i < s[t_w + (i+1)t_p > T_w + it_r]$ (theorem case 4b), $\exists i, 0 \leq i < s[T_f^p(i) > T_f^r(i-1)]$ and $T_A = t_w + T_p + t_r$ (Fig. 9c). □

Theorem 3. *The completion time using strategy A is the minimum possible completion time for every combination of t_w, t_p, and t_r.*



(a) case 1b

(b) case 1c

(c) case 4b

Fig. 9. Strategy A, $t_w < t_p$, $T_w - t_w > t_p$, $s = 4$ (cases 1b, 1c, and 4b).

Proof. First, we obtain a tighter lower bound L than the bound $t_w + \max\{(s-1)(t_w+t_r), s*t_p\}$ provided at the beginning of this section. Since, writes and reads are done serially on the same I/O channel, $L \geq T_w + T_r$. Since, $T_f^w(s-1) \geq T_w$ for every strategy, the processing of the last segment cannot begin before T_w. Hence, $L \geq T_w + t_p + t_r$. Since the processing of the first segment cannot beginning before t_w, the read of the first segment's output cannot complete before $t_w + t_p + t_r$. The remaining reads require $(s-1)t_r$ time and are done after the first read completes. So, the last read cannot complete before $t_w + t_p + T_R$. Hence, $L \geq t_w + t_p + t_R$. Also, since the processing of the first segment cannot begin before t_w, $T_f^p(s-1) \geq t_w + T_p$. Hence, $L \geq t_w + T_p + t_r$. Combining all of these bounds on L, we get $L \geq \max\{T_w + T_r, T_w + t_p + t_r, t_w + t_p + T_r, t_w + T_p + t_r\}$.

From Theorem 2, we see that, in all cases, T_A equals one of the expressions $T_w + T_r$, $T_w + t_p + t_r$, $t_w + t_p + T_r$, $t_w + T_p + t_r$. Hence, a tight lower bound on the completion time of every strategy is $L = \max\{T_w + T_r, T_w + t_p + t_r, t_w + t_p + T_r, t_w + T_p + t_r\}$. Strategy A achieves this tight lower bound (Theorem 2) and so obtains the minimum completion time possible. □

Theorem 4. *When $s > 1$, the completion time T_B of strategy B is*

$$T_B = t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w + t_r, t_p\} + \max\{t_p, t_r\} + t_r.$$

Proof. When the `for` loop index $i = 1$, the read within the loop begins at $t_w + \max\{t_w, t_p\}$. For $2 \leq i < s$, this read begins at $t_w + \max\{t_w, t_p\} + (i-1)\max\{t_w + t_r, t_p\}$. So, the final read, which is outside the loop, begins at $t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w + t_r, t_p\} + \max\{t_p, t_r\}$ and completes t_r units later. Hence,

$$T_B = t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w + t_r, t_p\} + \max\{t_p, t_r\} + t_r.$$

□

Theorem 5. *Strategy B does not guarantee minimum completion time.*

Proof. First, we consider two cases when T_B equals the tight lower bound L of Theorem 3. The first, is when $t_p = \min\{t_w, t_p, t_r\}$. Now, $T_B = T_W + T_r = L$. The second case is when $t_p \geq t_w + t_r$. Now, from Theorem 4, we obtain $T_B = t_w + T_p + t_r = L$. When, $s = 3$ and

$$t_w > t_p > t_r > 0, T_B = T_w + (s-1)t_r + t_p = T_w + T_r + t_p - t_r = T_w + t_p + 2t_r.$$

When strategy A is used with this data, we are either in case 1a of Theorem 2 and $T_A = T_w + T_r = L < T_w + T_r + t_p - t_r = T_B$ or we are in case 2 of Theorem 2 and $T_A = T_w + t_p + t_r = L < T_w + t_p + 2t_r = T_B$. □

The next theorem shows that the completion time when using strategy B is less than 13.33 percent more than when strategy A is used.

Theorem 6. $\frac{T_B - T_A}{T_A} = 0$ *when* $s \leq 2$ *and* $\frac{T_B - T_A}{T_A} < \frac{s-2}{s^2-1} \leq 2/15$ *when* $s > 2$.

Proof. We consider five cases that together exhaust all possibilities for the relationship among t_w, t_p, and t_r:

1. $t_w \geq t_p \wedge t_r \geq t_p$. From Theorems 2 (case 1a applies as $t_p \leq t_r \leq (s-1)t_r = T_r - t_r$) and 4, it follows that $T_A = T_B = T_w + T_r$. So, $(T_B - T_A)/T_A = 0$.
2. $t_w \geq t_p \wedge t_r < t_p$. From Theorem 4, $T_B = T_w + T_r + t_p - t_r$. We may be in either case 1a or case 2 of Theorem 2. If we are in case 1a, then $T_A = T_w + T_r$ and $T_r - t_r = (s-1)t_r \geq t_p$. So, $t_r \geq t_p/(s-1)$. (Note that since we also have $t_r < t_p$, s must be at least 3 for case 1a to apply.) Therefore, $t_w + t_r \geq (1 + 1/(s-1))t_p = \frac{s}{s-1}t_p$. Hence, for $s > 2$,

$$\frac{T_B - T_A}{T_A} = \frac{t_p - t_r}{s(t_w + t_r)} \leq \frac{t_p(1 - \frac{1}{s-1})}{\frac{s^2}{s-1}t_p} = \frac{s-2}{s^2} < \frac{s-2}{s^2-1}.$$

If we are in case 2 of Theorem 2, then $T_A = T_w + t_p + t_r$ and $t_w \geq t_p > T_r - t_r = (s-1)t_r$. So,

$$\frac{T_B - T_A}{T_A} = \frac{(s-2)t_r}{st_w + t_p + t_r}.$$

The right-hand side equals 0 when $s = 2$ and for $s > 2$, the right-hand side is

$$< \frac{(s-2)t_r}{s(s-1)t_r + (s-1)t_r + t_r} = \frac{s-2}{s^2} < \frac{s-2}{s^2-1}.$$

3. $t_w < t_p \wedge t_r \geq t_p$. Now, $T_B = T_w + T_r + t_p - t_w$. For strategy A, we are in case 1b, 3, or 4a of Theorem 2. If we are in case 1b, $T_A = T_w + T_r = s(t_w + t_r)$, $t_r \geq t_p$, and $(s-1)t_w = T_w - t_w > t_p$. (Note that since we also have $t_w < t_p$, s must be at least 3 for case 1b to apply.) So for $s > 2$,

$$\frac{T_B - T_A}{T_A} = \frac{t_p - t_w}{s(t_w + t_r)} < \frac{t_p\left(1 - \frac{1}{s-1}\right)}{s\left(\frac{t_p}{s-1} + t_p\right)} = \frac{s-2}{s^2} < \frac{s-2}{s^2-1}.$$

When case 3 of Theorem 2 applies, $T_A = t_w + t_p + T_r$ and $(s-1)t_w \leq t_p < t_r$. So,

$$\frac{T_B - T_A}{T_A} = \frac{(s-2)t_w}{t_w + t_p + st_r}.$$

The right-hand side equals 0 when $s = 2$ and for $s > 2$, the right-hand side is

$$\leq \frac{(s-2)t_w}{s(t_w + t_r)} < \frac{s-2}{s^2} < \frac{s-2}{s^2-1}.$$

When case 4a of Theorem 2 applies, $T_A = t_w + T_p + t_r$, $(s-1)t_w = T_w - t_w \leq t_p$, and $t_r = t_p$. So, $T_A = t_w + (s+1)t_p \geq t_w + (s+1)(s-1)t_w = s^2 t_w$ and $T_B - T_A = (s-1)t_w + (s+1)t_r - t_w - (s+1)t_r = (s-2)t_w$. Therefore,

$$\frac{T_B - T_A}{T_A} \leq \frac{(s-2)t_w}{s^2 t_w} = \frac{s-2}{s^2} < \frac{s-2}{s^2-1}.$$

4. $t_w < t_p \wedge t_r < t_p \wedge t_w + t_r \geq t_p$. Now, $T_B = T_w + T_r + 2t_p - t_w - t_r$. For strategy A, we are in case 1c, 4a or 4b of Theorem 2.
If we are in case 1c, $T_A = T_w + T_r$ and

$$t_w + st_p \leq st_w + (s-1)t_r.$$

So, $(s-1)(t_w + t_r) \geq st_p$ or $t_w + t_r \geq \frac{s}{s-1}t_p$. Hence,

$$\frac{T_B - T_A}{T_A} = \frac{2t_p - t_w - t_r}{s(t_w + t_r)} \leq \frac{2 - \frac{s}{s-1}}{\frac{s^2}{s-1}} = \frac{s-2}{s^2}.$$

The right-hand side is 0 when $s = 2$ and is $< \frac{s-2}{s^2-1}$ when $s > 2$.

For both cases 4a and 4b, $T_A = t_w + T_p + t_r$. When case 4a applies, $(s-1)t_w \leq t_p$. Since, $t_w + t_r \geq t_p$,

$$t_r \geq t_p - t_w \geq t_p\left(1 - \frac{1}{s-1}\right) = \frac{s-2}{s-1}t_p \geq (s-2)t_w.$$

Hence,

$$\frac{T_B - T_A}{T_A} = \frac{(s-2)(t_w + t_r - t_p)}{t_w + st_p + t_r}.$$

The right-hand side equals 0 when $s = 2$ and for $s > 2$, the right-hand side is

$$< \frac{(s-2)t_w}{t_w + s(s-1)t_w + (s-2)t_w} = \frac{s-2}{s^2-1}.$$

When case 4b applies, it follows from $t_p > t_r$ and the conditions for case 4b that $t_w + st_p > T_w + (s-1)t_r = st_w + (s-1)t_r$. So,

$$st_p > (s-1)(t_w + t_r)$$

and $t_w + t_r < \frac{s}{s-1}t_p$. Hence,

$$t_w + t_r - t_p < t_p/(s-1).$$

From this inequality and $t_w + t_r \geq t_p$, we get

$$\frac{T_B - T_A}{T_A} = \frac{(s-2)(t_w + t_r - t_p)}{t_w + st_p + t_r}.$$

The right-hand side equals 0 when $s = 2$ and for $s > 2$, the right-hand side is

$$< \frac{\frac{s-2}{s-1}t_p}{t_w + st_p + t_r} \leq \frac{\frac{s-2}{s-1}t_p}{(s+1)t_p} = \frac{s-2}{s^2-1}.$$

5. $t_w < t_p \wedge t_r < t_p \wedge t_w + t_r < t_p$. Now, $T_B = t_w + T_p + t_r = T_A$. Only cases 1c and 4a of Theorem 2 are possible when $t_w < t_p \wedge t_r < t_p$. However, since we also have $t_w + t_r < t_p$, $(s-1)t_p > (s-1)t_w + (s-1)t_r$. So, $t_w + (s-1)t_p > T_w + (s-1)t_r$. Hence, $t_w + st_p > T_w + (s-1)t_r$. Therefore, case 1c does not apply and $T_A = t_w + T_p + t_r = T_B$. So, $(T_B - T_A)/T_A = 0$. □

To see that the bound of Theorem 6 is quite tight, consider the instance $s = 4$, $t_r = t_p = 3$, and $t_w = 1$. For this instance, $T_A = t_w + T_p + t_r = 16$ (case 4a) and $T_B = 18$.

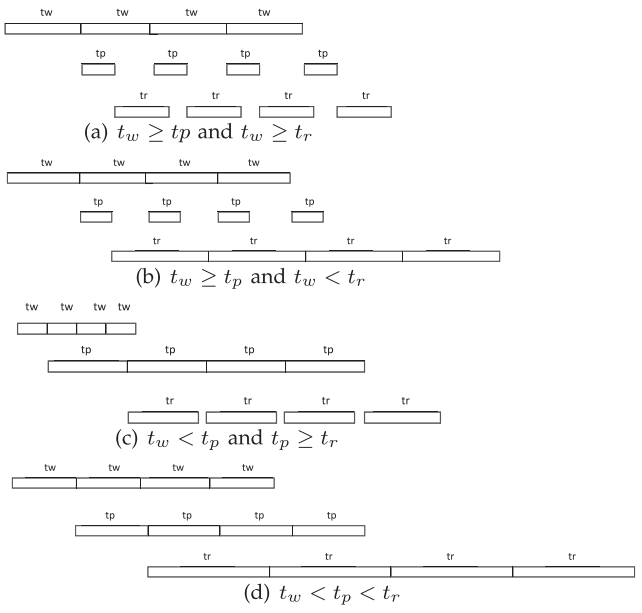


Fig. 10. Strategy A, enhanced GPU, $s = 4$.

So, $(T_A - T_B)/T_A = 1/8$. The instance falls into case 3 (subcase 4a of Theorem 2) of Theorem 6 for which we have shown

$$\frac{T_B - T_A}{T_A} \leq \frac{s-2}{s^2} \leq \frac{1}{8}.$$

Strategy B becomes more competitive with strategy A as the number of segments s increases. For example, when $s = 20$, $(T_B - T_A)/T_A < 18/399 = 0.045$.

4.3 Completion Time-Two I/O Channels

In this section, we analyze the completion times of strategies A and B under the assumption that we are using GPUs such as the Fermi that have two I/O channels/buses between the host CPU and the GPU and the CPU has a dual-port memory that supports simultaneous reads and writes. In this case, the writing of an input data segment to device memory can be overlapped with the reading of an output data segment from device memory. When $s = 1$, GPUs with two I/O channels are unable to perform any better than the original GPU and $T_A = T_B = t_w + t_p + t_r$. Theorems 7, 8, 9, 10, and 11 are the enhanced GPU analogs of Theorems 2, 3, 4, and 5 for the case $s > 1$. We refer to GPUs with two I/O channels as *enhanced GPUs*.

Theorem 7. *When $s > 1$, the completion time, T_A, of strategy A for the enhanced GPU model is*

$$T_A = \begin{cases} T_w + t_p + t_r & t_w \geq t_p \wedge t_w \geq t_r \\ t_w + T_p + t_r & t_w < t_p \wedge t_p \geq t_r \\ t_w + t_p + T_r & \text{otherwise.} \end{cases}$$

Proof. As for the original GPU model, $T_s^w(i) = i * t_w$, $0 \leq i < s$. When $t_w \geq t_p$, $T_s^p(i) = T_f^w(i) = (i+1) * t_w$ and $T_f^p(i) = T_f^w(i) + t_p = (i+1) * t_w + t_p$. Fig. 10a shows the schedule of events when $s = 4$, $t_w \geq t_p$, and $t_w \geq t_r$. Since $t_w \geq t_r$, $T_s^r(i) = T_f^p(i) = (i+1) * t_w + t_p$, $0 \leq i < s$. So, $T_s^r(s-1) = s * t_w + t_p$ and $T_A = T_f^r(s-1) = T_w + t_p + t_r$.

Fig. 10b shows the schedule of events when $s = 4$, $t_w \geq t_p$ and $t_w < t_r$. Since $t_w < t_r$, $T_s^r(i) = T_f^r(i-1) = T_s^r(i-1) + t_r$, $0 < i < s$. Since $T_f^r(0) = t_w + t_p + t_r$, $T_s^r(s-1) = t_w + t_p + (s-1)t_r$ and $T_A = T_f^r(s-1) = t_w + t_p + T_r$.

When $t_w < t_p$ and $t_p \geq t_r$, $T_s^p(i) = t_w + i * t_p$, $0 \leq i < s$, and $T_s^r(i) = T_f^p(i)$, $0 \leq i < s$ (Fig. 10c). So, $T_A = t_w + s * t_p + t_r = t_w + T_p + t_r$.

The final case to consider is $t_w < t_p < t_r$ (Fig. 10d). Now, $T_s^p(i) = t_w + i * t_p$, $0 \leq i < s$, and $T_s^r(i) = t_w + t_p + i * t_r$. So, $T_A = t_w + t_p + T_r$. □

Theorem 8. *For the enhanced GPU model, the completion time using strategy A is the minimum possible completion time for every combination of t_w, t_p, and t_r.*

Proof. The earliest time the processing of the last segment can begin is T_w. So, $T_f^p(s-1) \geq T_w + t_p$. So, the completion time L of every strategy is at least $T_w + t_p + t_r$. Further, the earliest time the read of the first output segment can begin is $t_w + t_p$. Following this earliest time, the read channel is needed for at least $s * t_r$ time to complete the reads. So, $L \geq t_w + t_p + T_r$. Also, since $T_s^p(0) = t_w$, the earliest time the processing of the last segment can begin is $t_w + (s-1)t_p$, Hence, $L \geq t_w + T_p + t_r$. Combining these lower bounds, we get

$$L \geq \max\{T_w + t_p + t_r, t_w + T_p + t_r, t_w + t_p + T_r\}.$$

Since T_A equals the derived lower bound, the lower bound is tight, $L = \max\{T_w + t_p + t_r, t_w + T_p + t_r, t_w + t_p + T_r\}$ and T_A is the minimum possible completion time. □

The next theorem shows that the optimal completion time using the original GPU model of Section 4.2 is at most twice that for the enhanced model of this section.

Theorem 9. *Let t_w, t_p, t_r, and s define a host-to-host instance. Let $C1$ and $C2$, respectively, be the completion time for an optimal host-to-host execution using the original and enhanced GPU models. $C1 \leq 2 * C2$ and this bound is tight.*

Proof. From the proofs of Theorems 3 and 8, it follows that $C1 = \max\{T_w + T_r, T_w + t_p + t_r, t_w + t_p + T_r, t_w + T_p + t_r\}$ and $C2 = \max\{T_w + t_p + t_r,\ t_w + t_p + T_r,\ t_w + T_p + t_r\}$. Hence, when $C1 \neq T_w + T_r$, $C1 = C2 \leq 2 * C2$. Assume that $C1 = T_w + T_r$. We consider two cases, $T_w \leq T_r$ and $T_w > T_r$. When, $T_w \leq T_r$, $C1 = T_w + T_r \leq 2 * T_r \leq 2 * C2$ (as $T_r \leq C2$). When, $T_w > T_r$, $C1 = T_w + T_r < 2 * T_w \leq 2 * C2$ (as $T_w \leq C2$).

To see that the bound is tight, suppose that $t_w = t_r$, and $t_p = \epsilon$. $C1 = 2 * s * t_w$ and $C2 = T_w + t_w + \epsilon = (s+1) * t_w + \epsilon$ (for ϵ sufficiently small and $s \geq 2$). So, $C1/C2 = (2 * s)/(s + 1 + \epsilon/t_w) \to 2$ as $s \to \infty$. □

Theorem 10. *When $s > 1$, the completion time T_B of strategy B for the enhanced GPU model is*

$$T_B = t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w, t_r, t_p\} + \max\{t_p, t_r\} + t_r.$$

Proof. When the `for` loop index $i = 1$, the read within this loop begins at $t_w + \max\{t_w, t_p\}$. For $2 \leq i < s$, this read begins at $t_w + \max\{t_w, t_r, t_p\}$. So, the final read, which is outside the loop, begins at $t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w, t_r, t_p\} + \max\{t_p, t_r\}$ and completes t_r units later. Hence,

$$T_B = t_w + \max\{t_w, t_p\} + (s-2)\max\{t_w, t_r, t_p\} + \max\{t_p, t_r\} + t_r.$$

□

Theorem 11. *Strategy B does not guarantee minimum completion time for the enhanced GPU model.*

Proof. First, we present a case when $T_B = L$. Suppose, $t_p \geq t_w$ and $t_p \geq t_r$. From Theorem 10, we obtain

$$T_B = t_w +_p +(s-2)t_p + t_p + t_r = t_w + T_p + t_r = L$$

However, when $t_w > t_r > t_p$, $T_B = t_w + t_w + (s-2)t_w + t_r + t_r = T_w + 2t_r > T_w + t_p + t_r = T_A = L$. □

Theorem 12 is the enhanced GPU analogue of Theorem 6.

Theorem 12. *For the enhanced GPU model, $(T_B - T_A)/T_A = 0$ when $s = 1$ and $(T_B - T_A)/T_B < 1/(s+1) \leq 1/3$ when $s > 1$. The bound is tight.*

Proof. It is easy to see that $T_B = T_A$ when $s = 1$. When $s > 1$, we consider five cases that together exhaust all possibilities for the relationship among t_w, t_p, and t_r:

1. $t_w = \max\{t_w, t_p, t_r\} \wedge t_p \geq t_r$.
   For this case, $T_B = T_w + t_p + t_r = T_A$.
2. $t_w = \max\{t_w, t_p, t_r\} \wedge t_p < t_r$.
   Now, $T_B = T_w + 2t_r$ and $T_A = T_w + t_p + t_r$. So,
   $$\frac{T_B - T_A}{T_A} = \frac{t_r - t_p}{st_w + t_p + t_r} < \frac{t_r}{st_w + t_r} \leq \frac{1}{s+1}.$$
   To see that this bound is tight, consider the s segment instance defined by $t_p = \epsilon$, and $t_w = t_r = 2$. For this instance, $T_B = 2s + 4$ and $T_A = 2s + \epsilon + 2$. So, $(T_B - T_A)/T_A = (2 - \epsilon)/(2s + \epsilon + 2)$, which $\to 1/(s+1)$ as $\epsilon \to 0$.
3. $t_p = \max\{t_w, t_p, t_r\} \wedge t_w < t_p$:
   Now, $T_B = t_w + T_p + t_r = T_A$.
4. $t_r = \max\{t_w, t_p, t_r\} \wedge t_w < t_r \wedge t_p < t_r \wedge t_w \geq t_p$
   For this case, $T_B = 2t_w + T_r$ and $T_A = t_w + t_p + T_r$. Hence,
   $$\frac{T_B - T_A}{T_A} = \frac{t_w - t_p}{t_w + t_p + st_r} < \frac{t_w}{t_w + st_r} \leq \frac{1}{s+1}.$$
   We note that this case is symmetric to case 2 above and the tightness of the bound may be established using a similar instance:
5. $t_r = \max\{t_w, t_p, t_r\} \wedge t_w < t_r \wedge t_p < t_r \wedge t_w < t_p$.
   Now, $T_B = t_w + t_p + T_r = T_A$.

□

5 Experimental Results

5.1 GPU-to-GPU

For all versions of our GPU-to-GPU CUDA code, we set $maxL = 17$, $T = 64$, and $S_{block} = 14{,}592$. Consequently, $S_{thread} = S_{block}/T = 228$ and $tWord = S_{thread}/4 = 57$. Note that since $tWord$ is odd, we will not have shared-memory bank conflicts (Theorem 1). We note that since our code is written using a 1D grid of blocks and since a grid dimension is required to be $< 65{,}536$ [10], our GPU-to-GPU code can handle at most 65,535 blocks. With the chosen block size, n must be less than 912 MB. For larger n, we can rewrite the code using a 2D indexing scheme for blocks.

TABLE 1
Runtime for AC Versions

Optimization Step	10MB	100MB	904MB
AC0	23ms	227ms	2158ms
AC1	12ms	118ms	1107ms
AC2	8ms	80ms	748ms
AC3	6ms	53ms	434ms
AC4	3ms	26ms	249ms

For our experiments, we used a pattern dictionary from [24] that has 33 patterns. The target search strings were extracted from a disk image and we used $n = 10$, 100, and 904 MB.

5.1.1 AC Algorithm

We evaluated the performance of the following versions of our GPU-to-GPU AC algorithm:

- AC0: This is Algorithm *basic* (Fig. 2) with the DFA stored in device memory.
- AC1: This differs from AC0 only in that the DFA is stored in texture memory.
- AC2: The AC1 code is enhanced so that each thread reads 16 characters at a time from device memory rather than 1. This reading is done using a variable of type `unint4`. The read data are stored in shared memory. The processing of the read data are done by reading it one character at a time from shared memory and writing the resulting state to device memory directly.
- AC3: The AC2 code is further enhanced so that threads cooperatively read data from device memory to shared memory as in Fig. 3 time. The read data are processed as in AC2.
- AC4: This is the AC3 code with deficiency D2 eliminated using a register array to save the input and cooperative writes as described in Section 3.2.2.

We experimented with a variant of AC3 in which data were read from shared memory as `uints`, the encoded four characters in a `uint` were extracted using shifts and masks, and DFA transitions done on these four characters. This variant took about 1 to 2 percent more time than AC3. Also, we considered variants of AC4 in which $tWord = 48$ and 56 and these, respectively, took approximately 14.78 and 7.8 percent more time that AC4. We do not report on these variants further.

Table 1 gives the runtime for each of our AC versions. As can be seen, the runtime decreases noticeably with each enhancement made to the code. Table 2 gives the speedup attained by each version relative to AC0 and Fig. 11 is a plot of this speedup. Simply relocating the DFA from device memory to texture memory as is done in AC1 results in a speedup of almost 2. Performing all of the enhancements yields a speedup of almost 8 when $n = 10$ MB and almost 9 when $n = 904$ MB.

TABLE 2
Speedup of AC1, AC2, AC3, and AC4 Relative to AC0

Optimization Step	10MB	100MB	904MB
AC0	1	1	1
AC1	1.93	1.92	1.95
AC2	2.80	2.83	2.89
AC3	4.11	4.26	4.97
AC4	7.71	8.58	8.68

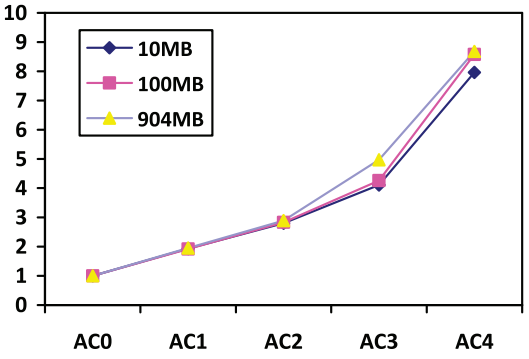


Fig. 11. Graphical representation of speedup relative to AC0.

5.1.2 Multipattern Boyer Moore Algorithm

For the multipattern Boyer Moore method, we considered only the versions mBM0 and mBM1 that correspond, respectively, to AC0 and AC1. In both mBM0 and mBM1, the bad character function and the $shift1$ and $shift2$ functions were stored in shared memory. Table 3 gives the runtimes for mBM0 and mBM1. Once again, relocating the reverse trie from device memory to texture memory resulted in a speedup of almost 2. Note that mBM1 takes between 7 and 10 percent more time than is taken by AC1. Since the multipattern Boyer-Moore algorithm has a somewhat more complex memory access pattern than used by AC, it is unlikely that the remaining code enhancements will be as effective as they were in the case of AC. So, we do not expect versions mBM2 through mBM4 to outperform their AC counterparts. So, we did not consider further refinements to mBM1.

5.1.3 Comparison with Multicore Computing on Host

For benchmarking purposes, we programmed also a multithreaded version of the AC algorithm and ran it on the quad-core Xeon host that our GPU is attached to. The multithreaded version replicated the AC DFA so that each thread had its own copy to work with. As shown in Table 4, for $n = 10$ MB and 100 MB we obtained best performance using eight threads while for $n = 500$ MB and 904 MB best performance was obtained using four threads. The 8-threads code delivered a speedup of 2.67 and 3.59, respectively, for $n = 10$ MB and 100 MB relative to the single-threaded code. For $n = 500$ and 904 MB, the speedup achieved by the 4-thread code was, respectively, 3.88 and 3.92!

AC4 offers speedups of 8.5, 9.2, and 9.5 relative to the single-thread CPU code for $n = 10$, 100, and 904 MB, respectively. The speedups relative to the best multithreaded quad-core codes were, respectively, 3.2, 2.6, and 2.4, respectively.

TABLE 3
Runtime for mBM Versions

Optimization Step	10MB	100MB	904MB
mBM0	25ms	252ms	2343ms
mBM1	13ms	127ms	1184ms

TABLE 4
Runtime for Multithreaded AC on Quad-Core Host

number of threads	10MB	speedup	100MB	speedup
1	24ms	1	243ms	1
2	14ms	1.81	126ms	1.94
4	11ms	2.17	69ms	3.54
8	9ms	2.67	68ms	3.59
16	11ms	2.30	68ms	3.58
number of threads	500MB	speedup	904MB	speedup
1	1238ms	1	2370ms	1
2	617ms	2.00	1206ms	1.96
4	319ms	3.88	605ms	3.92
8	367ms	3.37	677ms	3.50
16	356ms	3.47	621ms	3.82

5.2 Host-to-Host

We used AC3 with the parameters stated in Section 5.1 to process each segment of data on the GPU. The target string to be searched was partitioned into equal size segments. As a result, the time to write a segment to device memory was (approximately) the same for all segments as was the time to process each segment in the GPU and to read the results back to host memory. So, the assumptions made in the analysis of Section 4.2 apply. From Theorem 3, we know that host-to-host strategy A will give optimal performance (independent of the relative values of t_w, t_p, and t_r) though at the expense of requiring as much device memory as needed to store the entire input and the entire output. However, strategy B, while more efficient on memory when the number of segments is more than 2, does not guarantee minimum runtime. The values of t_w, t_p, and t_r for a segment of size 10 MB were determined to be 1.87, 2.73, and 3.63 ms, respectively. So, $t_w < t_p < t_r$ and this relative order will not change as we increase the segment size. When the number of segments is more than 2, we are in case 1b of strategy A. So, $T_A = T_w + T_r$. For strategy B, $T_B = T_w + T_r + t_p - t_w$, and strategy B is suboptimal. Strategy B is expected to take $t_p - t_w = 0.86$ ms more time than taken by strategy A when the segment size is 10 MB. Since t_w, t_r, and t_w scale roughly linearly with segment size, strategy B will be slower by about 8.6 ms when the segment size is 100 MB and by 77.7 ms when the segment size is 904 MB. Unless the value of n is sufficiently large to make strategy A infeasible because of insufficient device memory, we should use strategy A. We experimented with strategy A and Table 5 gives the time taken when $n = 500$ and 904 MB using a different number of segments. This figure also gives the speedup obtained by host-to-host strategy A relative to doing the multipattern search on the quad-core host using four threads (note that four threads give the fastest quad-core performance for the chosen values of n). Although the GPU delivers no speedup relative to our quad-core host, the speedup could be quite substantial when the GPU is a slave of a much slower host. In fact, when operating as a slave of a single-core host running at the same clock-rate as our Xeon host, the CPU times would be about the same as for our single-threaded version and the GPU host-to-host code would deliver a speedup of 3.1 when $n = 904$ MB and the number of segments is 1.

TABLE 5
Runtime for Strategy A Host-to-Host Code

segments	segment size	GPU	quadcore	speedup
100	9.04MB	817ms	605ms	0.74
10	90.4B	786ms	605ms	0.77
2	452MB	789ms	605ms	0.77
1	904MB	770ms	605ms	0.78
50	10MB	413ms	319ms	0.82
10	50MB	388ms	319ms	0.82
5	100MB	385ms	319ms	0.83
1	500MB	396ms	319ms	0.81

6 Conclusion

AC and mBM adaptations for the host-to-host and GPU-to-GPU cases were considered. For the host-to-host case, we suggest two strategies to communicate data between the host and GPU and showed that while strategy A was optimal with respect to runtime (under suitable assumptions), strategy B required lees device memory (when the number of segments is more than 2). Experiments show that the GPU-to-GPU adaptation of AC achieves speedups between 8.5 and 9.5 relative to a single-thread CPU code and speedups between 2.4 and 3.2 relative to a multi-threaded code that uses all cores of our quad-core host. For the host-to-host case, the GPU adaptation achieves a speedup of 3.1 relative to a single-thread code running on the host. However, for this case, a multithreaded code running on the quad core is faster. Of course, performance relative to the host is quite dependent on the speed of the host and using a slower or faster host with fewer or more cores will change the relative performance values.

Acknowledgments

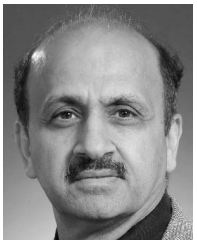
This research was supported, in part, by the US National Science Foundation under grants 0829916 and CNS-0963812.

References

[1] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM*, vol. 18, no. 6, pp. 333-340, 1975.
[2] R. Baeza-Yates, "Improved String Searching," *Software-Practice and Experience*, vol. 19, pp. 257-271, 1989.
[3] R. Baeza-Yates and G. Gonnet, "A New Approach to Text Searching," *Comm. ACM*, vol. 35, no. 10, pp. 74-82, 1992.
[4] R. Boyer and J. Moore, "A Fast String Searching Algorithm," *Comm. ACM*, vol. 20, no. 10, pp. 262-272, 1977.
[5] S. Che et al., "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *J. Parallel and Distributed Computing*, vol. 68, pp. 1370-1380, 2008.
[6] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," *Proc. Sixth Int'l Colloquium Automata, Languages and Programming (ICALP)*, pp. 118-132, 1979.
[7] M. Crochemore et al., "Speeding up Two String Matching Algorithms," *Algorithmica*, vol. 12, pp. 247-267, 1994.
[8] M. Crochemore et al., "Fast Practical Multi-Pattern Matching," *Information Processing Letters*, vol. 71, pp. 107-113, 1999.
[9] M. Crochemore and W. Rytter, *Text Algorithms*. Oxford Univ. Press, 1994.
[10] http://developer.nvidia.com/object/gpucomputing.html, 2012.
[11] M. Fisk and G. Varghese, "Applying Fast String Matching to Intrusion Detection," Los Alamos Nat'l Lab NM, 2002.
[12] Z. Galil, "On Improving the Worst Case Running Time of Boyer-Moore String Matching Algorithm," *Proc. Int'l Colloquium Automata, Languages and Programming (ICALP)*, 1978.
[13] N. Horspool, "Practical Fast Searching in Strings," *Software-Practice and Experience*, vol. 10, pp. 501-506, 1980.
[14] N. Huang, H. Hung, and S. Lai, "A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems," *Proc. 22nd Int'l Conf. Advanced Information Networking and Applications*, 2008.

[15] N. Jacob and C. Brodley, "Offloading IDS Computation to the GPU," *Proc. 22nd Ann. Computer Security Applications Conf.,* 2006.
[16] D.E. Knuth, J.H. Morris Jr., and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing,* vol. 6, 323-350, 1977.
[17] C. Lin et al., "Accelerating String Matching Using Multi-Threaded Algorithm on GPU," *Proc. IEEE Globecom,* 2010.
[18] L. Marziale, G. Richard III, and V. Roussev, "Massive Threading: Using GPUs to Increase the Performance of Digit Forensics Tools," *Science Direct,* vol. 4, pp. 73-81, 2007.
[19] G. Navarro and K. Frederiksson, "Average Complexity of Exact and Approximate Multiple String Matching," *Theoretical Computer Science,* vol. 321, pp. 283-290, 2004.
[20] A. Pal and N. Memon, "The Evolution of File Carving," *IEEE Signal Processing Magazine,* vol. 26, no. 2, pp. 59-72, Mar. 2009.
[21] "PFAC: A Library for String Matching on NVIDIA GPUs," http://code.google.com/p/pfac/, 2011.
[22] G. Richard III and V. Roussev, "Scalpel: A Frugal, High Performance File Carver," *Proc. Digital Forensics Research Workshop,* 2005.
[23] S. Sahni, "Scheduling Master-Slave Multiprocessor Systems," *IEEE Trans. Computers,* vol. 45, no. 10, pp. 1195-1199, Oct. 1996.
[24] http://www.digitalforensicssolutions.com/Scalpel/, 2012.
[25] D. Scarpazza, O. Villa, and F. Petrini, "Peak-Performance DFA-Based String Matching on the Cell Processor," *Proc. Int'l Workshop System Management Techniques, Processes, and Services,* 2007.
[26] D. Scarpazza, O. Villa, and F. Petrini, "Accelerating Real-Time String Searching with Multicore Processors," *Computer,* vol. 41, no. 4, pp. 42-50, Apr. 2008.
[27] R. Smith et al., "Evaluating GPUs for Network Packet Signature Matching," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* 2009.
[28] http://www.snort.org/dl, 2012.
[29] http://www.lostcircuits.com/graphics, 2012.
[30] A. Tumeo, S. Seechi, and O. Villa, "Experiences with String Matching on the Fermi Architecture," *Proc. 24th Int'l Conf. Architecture of Computing Systems (ARCS),* pp. 26-37, 2011.
[31] G. Vasiliadis et al., "Regular Expression Matching on Graphics Hardware for Intrusion Detection," *Proc. 12th Int'l Symp. Recent Advances in Intrusion Detection (ISRAID),* 2009.
[32] Y. Won and S. Sahni, "A Balanced Bin Sort for Hypercube Multicomputers," *J. Supercomputing,* vol. 2, pp. 435-448, 1988.
[33] Y. Won and S. Sahni, "Hypercube-to-Host Sorting," *J. Supercomputing,* vol. 3, pp. 41-61, 1989.
[34] Y. Won and S. Sahni, "Host-to-Hypercube Sorting," *Computer Systems: Science and Eng.,* vol. 4, no. 3, pp. 161-168, 1989.
[35] S. Wu and U. Manber, "Agrep-A Fast Algorithm for Multi-Pattern Searching," technical report, Univ. of Arizona, 1994.
[36] A. Yao, "The Complexity of Pattern Matching for a Random String," *SIAM J. Computing,* vol. 8, pp. 368-387, 1979.
[37] X. Zha, D. Scarpazza, and S. Sahni, "Highly Compressed Multi-Pattern String Matching on the Cell Broadband Engine," *Proc. IEEE Symp. Computers and Comm. (ISCC),* 2011.
[38] X. Zha and S. Sahni, "Fast in-Place File Carving for Digital Forensics," *Proc. e-Forensics,* pp. 141-158, 2010.
[39] X. Zha and S. Sahni, "Multipattern String Matching on a GPU," *Proc. IEEE Symp. Computers and Comm. (ISCC),* 2011.

Xinyan Zha received the PhD degree from the Computer and Information Science and Engineering Department at the University of Florida in the summer of 2010. She worked under the supervision of Dr. Sartaj Sahni. She received the Best Student Paper Award from IEEE Symposium on Computers and Communications in 2011. Her research interests include data structures and algorithms, network intrusion detection systems, and parallel computing.



Sartaj Sahni is a distinguished professor and chair of Computer and Information Sciences and Engineering at the University of Florida. He is also a member of the European Academy of Sciences and a distinguished alumnus of the Indian Institute of Technology, Kanpur. He received the IEEE Computer Society Taylor L. Booth Education Award "for contributions to Computer Science and Engineering education in the areas of data structures, algorithms, and parallel algorithms," in 1997, the IEEE Computer Society W. Wallace McDowell Award "for contributions to the theory of NP-hard and NP-complete problems," in 2003, and the 2003 ACM Karl Karlstrom Outstanding Educator Award for "outstanding contributions to computing education through inspired teaching, development of courses and curricula for distance education, contributions to professional societies, and authoring significant textbooks in several areas including discrete mathematics, data structures, algorithms, and parallel and distributed computing." He has published more than 300 research papers and written 15 texts. His research publications are on the design and analysis of efficient algorithms, parallel computing, interconnection networks, design automation, and medical algorithms. He is a fellow of the IEEE, ACM, AAAS, and Minnesota Supercomputer Institute.