Multipattern String Matching On A GPU

Xinyan Zha and Sartaj Sahni
Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
Email: {xzha, sahni}@cise.ufl.edu

***Abstract*—We develop GPU adaptations of the Aho-Corasick string matching algorithm for the two cases GPU-to-GPU and host-to-host. 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 run time while the other requires less device memory. Experiments conducted on an NVIDIA Tesla GT200 GPU that has 240 cores running off of a Xeon 2.8GHz 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.**

Keywords: Multipattern string matching, Aho-Corasick, GPU, CUDA.

I. 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 [13] has a dictionary of several thousand patterns that are matched against the contents of Internet packets and the open-source file carver Scalpel [9] 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 [13] employs the Aho-Corasick [1] multipattern search method while Scalpel [9] uses the Boyer-Moore single pattern search algorithm [2]. Since Scalpel uses a single pattern search algorithm, its run time 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 run time 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. [10], [11] port the deterministic finite automata version of the Aho-Corasick method to the IBM Cell Broadband Engine (CBE) while Zha et al. [16] port a compressed form of the non-deterministic finite automata version of the Aho-Corasick method to the CBE. Jacob et al. [5] port Snort to a GPU. However, in their port, they replace the Aho-Corasick search method employed by Snort with the Knuth-Morris-Pratt [6] single-pattern matching algorithm. Specifically, they search for 16 different patterns in a packet in parallel employing 16 GPU cores. Huang et al. [4] do network intrusion detection on a GPU based on the multipattern search algorithm of Wu and Manber [15]. Smith et al. [12] use deterministic finite automata and extended deterministic finite automata to do regular expression matching on a GPU for intrusion detection applications. Marziale et al. [7] propose the use of GPUs and massive parallelism for in-place file carving. However, Zha and Sahni [17] 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 Aho-Corasick multipattern string matching algorithm through the use of a GPU. In this paper, we address two cases for the placement of the input and output–GPU-to-GPU (the target string resides in the device memory and the results are to be left in this memory) and host-to-host (the target string is in the CPU memory and the results of the matching are to be left in the CPU memory). In both cases, we assume that the pattern data structure is precomputed and stored in the GPU. Although we researched GPU adaptations of the Boyer-Moore multistring matching algorithm as well, these adaptations did not perform as well as our GPU adapatations of the Aho-Corasick algorithm. So, we do not report on the Boyer-Moore adaptations here.

The remainder of this paper is organized as follows. Sec-

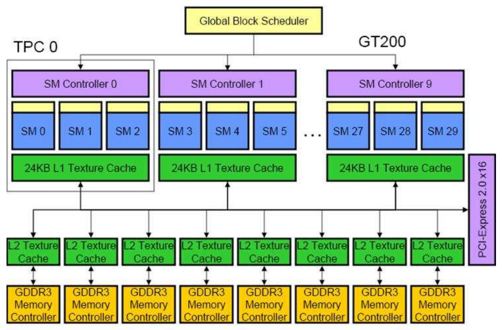

Fig. 1. NVIDIA GT200 Architecture [14]

tion II introduces the NVIDIA Tesla architecture. In Section III we describe the Aho-Corasick algorithm. Sections IV and V describe our GPU adaptation for the GPU-to-GPU and host-to-host cases. Section VI discusses our experimental results and we conclude in Section VII.

II. The NVIDIA Tesla Architecture

Figure 1 gives the architecture of the NVIDIA GT200 Tesla GPU, which is an example of NVIDIA's general purpose parallel computing architecture CUDA (Compute Unified Driver Architecture) [3]. This GPU comprises 240 scalar processors (SP) or cores that are organized into 30 streaming multiprocessors (SM) each comprised of 8 SPs. Each SM has 16KB of on-chip shared memory, 16384 32-bit registers, and constant and texture cache. Each SM supports up to 1024 active threads. There also is 4GB of global or device memory that is accessible to all 240 SPs. The Tesla, like other GPUs, operates as a slave processor to an attached host. In our experimental setup, the host is a 2.8GHz Xeon quad-core processor with 16GB of memory.

A CUDA program typically is a C program written for the host. C extensions supported by the CUDA programming environment allow the host to send and receive data to/from the GPU's device memory as well as to invoke C functions (called kernels) that run on the GPU cores. The GPU programming model is Single Instruction Multiple Thread (SIMT). When a kernel is invoked, the user must specify the number of threads to be invoked. This is done by specifying explicitly the number of thread blocks and the number of threads per block. CUDA further organizes the threads of a block into warps of 32 threads each, each block of threads is assigned to a single SM, and thread warps are executed synchronously on SMs. While thread divergence within a warp is permitted, when the threads of a warp diverge, the divergent paths are executed serially until they converge.

A CUDA kernel may access different types of memory with each having different capacity, latency and caching properties. We summarize the memory hierarchy below.

- Device memory: 4GB of device memory are available. This memory can be read and written directly by all threads. However, device memory access entails a high latency (400 to 600 clock cycles). The thread scheduler attempts to hide this latency by scheduling arithmetics that are ready to be performed while waiting for the access to device memory to complete [3]. Device memory is not cached.
- Constant memory: Constant memory is read-only memory space that is shared by all threads. Constant memory is cached and is limited to 64KB.
- Shared memory: Each SM has 16KB of shared memory. Shared memory is divided into 16 banks of 32-bit words. When there are no bank conflicts, the threads of a warp can access shared memory as fast as they can access registers [3].
- Texture memory: Texture memory, like constant memory, is read-only memory space that is accessible to all threads using device functions called texture fetches. Texture memory is initialized at the host side and is read at the device side. Texture memory is cached.
- Pinned memory (also known as Page-Locked Host Memory): This is part of the host memory. Data transfer between pinned and device memory is faster than between pageable host memory and device memory. Also, this data transfer can be done concurrent with kernel execution. However, since allocating part of the host memory as pinned "reduces the amount of physical memory available to the operating system for paging, allocating too much page-locked memory reduces overall system performance" [3].

III. The Aho-Corasick Algorithm

There are two versions–nondeterministic and deterministic–of the Aho-Corasick (AC) [1] multipattern matching algorithm. We use the deterministic version in our work as it makes half as many state transitions as made by the non-deterministic version. In the deterministic version (DFA), each state has a transition pointer for every character in the alphabet as well as a list of matched patterns. Aho and Corasick [1] show how to compute the transition pointers. The number of state transitions made by a DFA when searching for matches in a string of length n is n.

Figure 2 gives the Aho-Corasick DFA for the patterns abcaabb, abcaabbcc, acb, acbccabb, ccabb, bccabc, and bbccabca drawn from the 3-letter alphabet {a,b,c}

IV. GPU-to-GPU

A. Strategy

The input to the multipattern matcher is a character array $input$ and the output is an array $output$ of states. Both arrays reside in device memory. $output[i]$ 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. If we assume that the number of states in the AC DFA is no more than 65536, a state can be encoded using two bytes and the size of the output array is twice that of the input array.

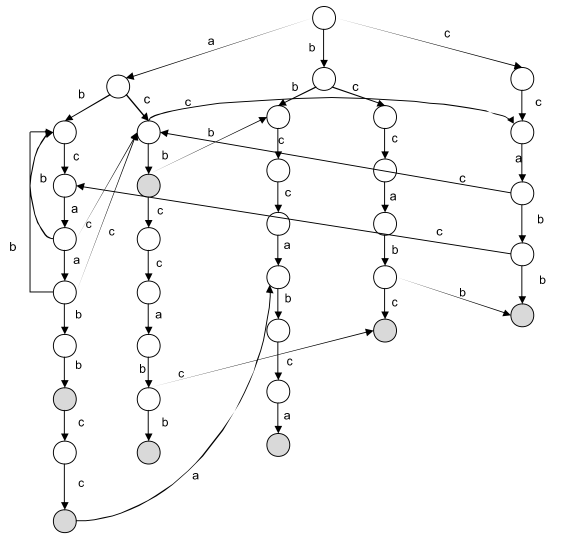

Fig. 2. Example Aho-Corasick DFA

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. 3. GPU-to-GPU notation

Our computational strategy is to partition the output array into blocks of size S_{block} (Figure 3 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). 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 $input[b*S_{block} + t*S_{thread} - maxL + 1 : b*S_{block} + t*S_{thread} + S_{thread} - 1]$. Figure 4 gives the pseudocode for a T-thread computation of block i of the output

```
**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. 4. Overall GPU-to-GPU strategy using AC

using the AC DFA. The variables used are self-explanatory and the correctness of the pseudocode follows from the preceding discussion.

The AC DFA resides in texture memory because texture memory is cached and is sufficiently large to accommodate the DFA. While shared and constant memories will result in better performance, neither is large enough to accommodate the DFA. 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 65536, each state transition of a DFA takes 2 bytes. So, a DFA with d states requires $512d$ bytes. In the 16KB shared memory that our Tesla has, we can store at best a 32-state DFA. The constant memory on the Tesla is 64KB. So, this can handle, at best, a 128-state DFA.

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 Figure 4 does, however, have deficiencies that are expected to result in non-optimal performance on a GPU. These deficiencies are listed 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–(a) 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. (b) 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} = 65536 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.

Analysis of Total Work

Using the GPU-to-GPU strategy of Figure 4, 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

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

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.

B. Addressing the Deficiencies

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

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

```
// 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. 5. T threads collectively read a block and save in shared memory

A variable `var` of type `uint4` is comprised of 4 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 4 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 8 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 Figure 5, 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%. Although 100% 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 2 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 4 unsigned integer components of a `uint4`.

Although the input scheme of Figure 5 succeeds in reading in the data utilizing 100% 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 mod 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 $(t * tWord)$ mod 16, $0 \le t < 16$. For our example, $tWord = 56$ and $(t * 56)$ mod $16 = 0$ when t is even and $(t * 56)$ mod $16 = 8$ when t is odd. Since each bank is accessed 8 times by the half warp, the reads by a half warp are serialized to 8 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, $(i * tWord)$ mod $16 \ne (jk)$ mod 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 $(i * tWord)$ mod $16 = (j * tWord)$ mod 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.

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. This reduction in block size increases the total work overhead significantly. We can avoid this increase in total work overhead by doing the following: (a) First, each thread processes the first $maxL - 1$ characters it is to process. The processing of these characters generates no output and so we need no memory to store output. (b) 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 $S_{block} \le 4 * 16384 = 64$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 $S_{block} \le 4864$, 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 non-integral 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 4 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 16KB shared memory, S_{thread} can be at most $16 * 1024/64 = 256$ 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%. 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.

V. Host-to-Host

Since the Tesla GPU supports 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. However, since there is only 1 I/O channel between the host and the GPU, time spent writing to the GPU cannot be overlapped with the time spent reading from the GPU. There are at least two ways to accomplish the overlap of I/O between host and device and GPU computation. In Strategy A (Figure 6), which is given in [3], we have three loops. The first loop asynchronously writes the input data to device memory in segments, the second processes each segment on the GPU,

```
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. 6. Host-to-host strategy A

```
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. 7. Host-to-host strategy B

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 Figure 6, 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. Figure 7 gives an alternative strategy that requires only sufficient device memory for 2 segments (2 input buffers IN0 and IN1 and two output buffers OUT0 and OUT1). We could, of course, couple strategies A and B to obtain a hybrid strategy.

We have analyzed the relative time performance of these two host-to-host strategies in [8]. Due to space limitations, we only summarize the results of this analysis here.

1) Strategy A gurantees the minimum possible completion

time while stratgey B does not. However, as noted earlier, strategy A requires more device memory than required by strategy B.

2) The completion time when strategy B is used is at most 13.33% more than when strategy A is used and this bound is tight.
3) If the GPU system is enhanced to have two I/O channels between the host and the GPU and the CPU has a dual port memory that supports simultaneous reads and writes, strategy A remains optimal and B remains suboptimal; the completion time using strategy B is at most 33% more than when strategy A is used and this bound is tight; and the stated enhancement of the GPU system results in at most a 50% reduction in completion time (this bound also is tight).

VI. Experimental Results

A. GPU-to-GPU

For all versions of our GPU-to-GPU CUDA code, we set $maxL = 17$, $T = 64$, and $S_{block} = 14592$. 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 1-dimensional grid of blocks and since a grid dimension is required to be < 65536 [3], our GPU-to-GPU code can handle at most 65535 blocks. With the chosen block size, n must be less than 912MB. For larger n, we can rewrite the code using a two-dimensional indexing scheme for blocks.

For our experiments, we used a pattern dictionary from [9] that has 33 patterns. The target search strings were extracted from a disk image and we used n = 10MB, 100MB, and 904MB.

1) Aho-Corasick Algorithm: We evaluated the performance of the following versions of our GPU-to-GPU AC algorithm:

AC0 This is Algorithm $basic$ (Figure 4) 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 is stored in shared memory. The processing of the read data is 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 Figure 5. time. The read data is 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 IV-B2.

We experimented with a variant of AC3 in which data was read from shared memory as `uint`s, the encoded 4 characters in a `uint` were extracted using shifts and masks, and DFA transitions done on these 4 characters. This variant took about 1% to 2% more time than AC3 and is not reported on further. Also, we considered variants of AC4 in which $tWord$ = 48 and 56 and these, respectively, took approximately 14.78% and 7.8% more time that AC4. We do not report on these variants further either.

Table I gives the run time for each of our AC versions. As can be seen, the run time decreases noticeably with each enhancement made to the code. Table II gives the speedup attained by each version relative to AC0 and Figure 8 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 = 10MB and almost 9 when n = 904MB.

2) 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. For n = 10MB and 100MB we obtained best performance using 8 threads while for n = 500MB and 904MB best performance was obtained using 4 threads. The 8-threads

TABLE I
Run time for AC versions

Optimization Step	10MB	100MB	904MB
AC0	22.92ms	227.12ms	2158.31ms
AC1	11.85ms	118.14ms	1106.75ms
AC2	8.19ms	80.34ms	747.73ms
AC3	5.57ms	53.33ms	434.03ms
AC4	2.88ms	26.48ms	248.71ms

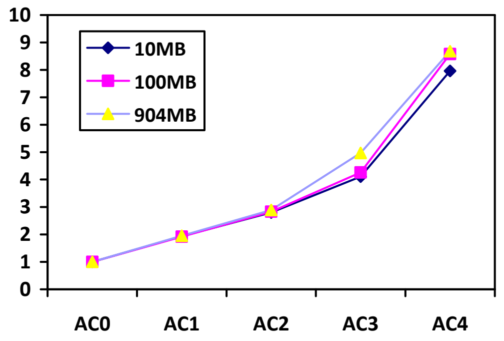

Fig. 8. Graphical representation of speedup relative to AC0

TABLE II
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

TABLE III
RUN TIME FOR MULTITHREADED AC ON QUAD-CORE HOST

number of threads	10MB	speedup	100MB	speedup
1	24.48ms	1	243.47ms	1
2	13.52ms	1.81	125.52ms	1.94
4	11.28ms	2.17	68.74ms	3.54
8	9.18ms	2.67	67.77ms	3.59
16	10.64ms	2.30	68.07ms	3.58
number of threads	500MB	speedup	904MB	speedup
1	1237.64ms	1	2369.85ms	1
2	617.44ms	2.00	1206.21ms	1.96
4	319.23ms	3.88	604.54ms	3.92
8	367.32ms	3.37	677.16ms	3.50
16	356.48ms	3.47	620.99ms	3.82

code delivered a speedup of 2.67 and 3.59, respectively, for n = 10MB and 100MB relative to the single-threaded code. For n = 500MB and 904MB, the speedup achieved by the 4-thread code was, respectively, 3.88 and 3.92, which is very close to the maximum speedup of 4 that a quad-core can deliver.

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

B. Host-to-Host

We used AC3 with the parameters stated in Section VI-A 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. From our analysis [8], we know that host-to-host strategy A will give optimal performance while, for the selected parameters, strategy B will not give optimal performance. So, we experimented only with strategy A. Table IV gives the time taken when n = 500MB and 904MB 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 4 threads (note that 4 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 = 904MB and 500MB and the number of segments is 1.

VII. CONCLUSION

We focus on multistring pattern matching using a GPU. AC 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 run time (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 multithreaded 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.

REFERENCES

[1] A. Aho and M. Corasick, Efficient string matching: An aid to bibliographic search, CACM, 18, 6, 1975, 333-340.

[2] R. Boyer and J. Moore, A fast string searching algorithm, *CACM*, 20, 10, 1977, 262-272.

[3] NVIDIA CUDA manual reference, http://developer.nvidia.com/object/gpucomputing.html

[4] N. Huang, H. Hung, S.Lai et al, A GPU-based Multiple-pattern Matching Algorithm for Network Intrusion Detection Systems, *The 22nd International COnference on Advanced Information Networking and Applications, 2008*

[5] N. Jacob, C.Brodley, Offloading IDS Computation to the GPU, *The 22nd Annual Computer Security Applications Conference, 2006*

[6] D.E.Knuth, J.H. Morris, Jr, and V.R.Pratt, *Fast pattern matching in strings*, SIAM J. Computing 6, 323-350, 1977.

[7] L. Marziale, G. Richard III, V. Roussev, Massive Threading: Using GPUs to increase the performance of digit forensics tools, *Science Direct, 2007*

[8] http://www.cise.ufl.edu/s̃ahni/papers/gpuMatching.pdf

[9] http://www.digitalforensicssolutions.com/Scalpel/

[10] D. Scarpazza, O. Villa, F. Petrini, Peak-Performance DFA-based String Matching on the Cell Processor, *Third IEEE/ACM Intl. Workshop on System Management Techniques, Processes, and Services, within IEEE/ACM Intl. Parallel and Distributed Processing Symposium 2007*

[11] D. Scarpazza, O.Villa, F.Petrini, Accelerating Real-Time String Searching with Multicore Processors, *IEEE Computer Society* , 2008.

[12] R. Smith, N. Goyal, J. Ormont et al. Evaluating GPUs for Network Packet Signature Matching, *International Symposium on Performance Analysis of Systems and Software*, 2009.

[13] http://www.snort.org/dl.

[14] NVIDA tesla architecture, http://www.lostcircuits.com/graphics.

[15] S. Wu and U. Manber, Agrep–a fast algorithm for multi-pattern searching, Technical Report, Department of Computer Science, University of Arizona, 1994.

[16] X. Zha, D. Scarpazza, and S. Sahni, Highly compressed multi-pattern string matching on the Cell Broadband Engine, University of Florida, 2009.

[17] X. Zha and S. Sahni, Fast in-place file carving for digital forensics, *e-Forensics*, LNICST, Springer, 2010.

TABLE IV
RUN TIME FOR STRATEGY A HOST-TO-HOST CODE

segments	segment size	GPU	quadcore	speedup
100	9.04MB	816.80ms	604.54ms	0.74
10	90.4MB	785.55ms	604.54ms	0.77
2	452MB	788.63ms	604.54ms	0.77
1	904MB	770.13ms	604.54ms	0.78
50	10MB	412.55ms	319.23ms	0.82
10	50MB	387.78ms	319.23ms	0.82
5	100MB	385.17ms	319.23ms	0.83
1	500MB	396.42ms	319.23ms	0.81