A Memory-Access-Efficient Implementation for Computing the Approximate String Matching Algorithm on GPUs

Lucas Saad Nogueira NUNES\textsuperscript{1}, Jacir Luiz BORDIM\textsuperscript{1}\textsuperscript{*}, Yasuaki ITO\textsuperscript{1}\textsuperscript{††}, and Koji NAKANO\textsuperscript{1}\textsuperscript{††}, Members

SUMMARY The closeness of a match is an important measure with a number of practical applications, including computational biology, signal processing and text retrieval. The approximate string matching (ASM) problem asks to find a substring of string \( Y \) of length \( n \) that is most similar to string \( X \) of length \( m \). It is well-known that the ASM can be solved by dynamic programming technique by computing a table of size \( m \times n \). The main contribution of this work is to present a memory-access-efficient implementation for computing the ASM on a GPU. The proposed GPU implementation relies on warp shuffle instructions which are used to accelerate the communication between threads without resorting to shared memory access. Despite the fact that \( O(mn) \) memory access operations are necessary to access all elements of a table with size \( n \times m \), the proposed implementation performs only \( O(m) \) memory access operations, where \( w \) is the warp size. Experimental results carried out on a GeForce GTX 980 GPU show that the proposed implementation, called \( w \)-SCAN, provides speed-up of over two fold in computing the ASM as compared to another prominent alternative.

\textbf{key words:} approximate string matching, edit distance, GPU, CUDA, shuffle instructions

1. Introduction

The task of finding strings that match a given pattern has a number of applications, ranging from natural text searching or when handling large DNA sequences\cite{2}, \cite{3}. Suppose that two strings, say \( X \) and \( Y \) of length \( m \) and \( n \), respectively, are given. The Approximate String Matching (ASM) is a task to find a substring in \( Y \) most similar to \( X \). The closeness of a match is measured in terms of the number of primitive operations necessary to convert a string into an exact match of the other. That is, the matching allows for a limited number of differences between the two strings. Approximate string matching algorithms are often classified into two main categories: online and offline\cite{4}. Offline solutions are characterized by the usage of indexing schemes to accelerate the matching task. That is, the indexing provides a persistent data structure to speed-up the ASM task. Despite of its benefits, building an appropriate indexing is a costly operation, requiring the text to be pre-processed in advance, which may be possible for certain applications\cite{4}. The above facts have fostered the quest for fast, online, ASM solutions. It is well-known that the ASM can be computed in \( O(mn) \) time using dynamic programming technique, which creates a table of size \( m \times n \) to store partial solutions\cite{5}. Approximate string matching variations, providing a reduced computing time has been proposed in the literature. In \cite{3}, it is shown that the computing time can be reduced when the task at hand requires listing substrings in \( Y \) with similarity of no more than \( k \). When more elaborated bit operations on words are allowed, Mayers\cite{6} showed that the ASM can be computed in \( O(mn/s) \), where \( s \) is the machine word size.

Aiming at accelerating the ASM task, GPU (Graphics Processing Unit) implementations have been proposed\cite{7}, \cite{8}. The GPU is a specialized circuit designed to accelerate computation for building and manipulating images\cite{9}. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called Compute Unified Device Architecture (CUDA)\cite{10}. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs.

NVIDIA GPUs have a number of streaming multiprocessors (SMP) that can execute multiple threads in parallel. CUDA uses different types of memories in the NVIDIA GPUs, of particular importance is the \textit{shared memory} and the \textit{global memory}\cite{10}. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid bank conflicts. To maximize the bandwidth between the GPU and the global memory, consecutive addresses must be accessed at the same time. Thus, CUDA threads should perform \textit{coalesced} access when they access the global memory. When no bank conflict occurs, the shared memory provides a much lower latency than uncached global memory.

The efficient use of the shared memory is the key element for CUDA developers to accelerate applications using GPUs. In \cite{11}, Nakano introduced the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) models, whose objective is to capture the essential features of the shared memory and the global memory of NVIDIA GPUs. The Hierarchical Memory Machine (HMM) model,
consisting of multiple DMMs and a single UMM, has been proposed in [12]. Using this model, an optimal algorithm for the ASM on the DMM and the HMM was presented in [7]. This algorithm, hereafter denoted as “O-ASM”, uses a matrix \( e \) of size \( 3 \times (m + 1) \) to store intermediate values while computing the approximate string matching. The O-ASM ensures that the shared memory accesses to matrix \( e \) are conflict-free. The O-ASM, however, requires a significant number of shared memory operations. Indeed, it needs \((3m \times (n + m - 1))\)-read and \((m \times n \times (n + m - 1))\)-write shared memory operations. Also, intuitively, to solve the ASM problem, \( O(mn) \) total memory access operations are necessary to access all elements in the table of size \( m \times n \). In this work, we explore an ingenious usage of warp shuffle operations, which support direct communication between threads in a warp without accessing the shared memory. This, in turn, allows to reduce the total number of shared memory access.

The main contribution of this work is to propose a parallel algorithm for computing the ASM on the GPU. The proposed algorithm, termed \( w \)-SCAN, explores the use of shuffle operations [10]. These operations become available on recent NVIDIA GPUs and are used in this work to speed-up the ASM computation. By properly managing shared memory accesses, \( w \)-SCAN reduces the amount of read/write operations while computing intermediate values of the ASM, allowing it to provide a better speed-up than that of obtained by O-ASM.

Table 1 shows the number of shared memory accesses taken by \( w \)-SCAN and O-ASM implementations. As can be observed from the table, \( w \)-SCAN requires \((\frac{2n}{w}) \times (n + m - 1))\)-read and \((\frac{m}{w}) \times (n + m - 1))\)-write shared memory accesses to compute the ASM, where \( w \) is the warp size, \( m \) and \( n \) are the length of strings \( X \) and \( Y \), respectively. Quite surprisingly, the total number of memory access operations is only \( O(\frac{mn}{w}) \), which is much smaller than the size of the table. The \( w \)-SCAN algorithm has been implemented on the GeForce GTX 980 GPU. Experimental results show that, while computing the ASM, the proposed \( w \)-SCAN implementation provides speed-up surpassing two times as compared to the O-ASM implementation.

The rest of this paper is organized as follows. Section 2 provides an overview of the approximate string matching problem and details some of the key aspects of the O-ASM algorithm. Section 3 presents the shuffle instructions considered in this work. In addition, this section provides a comparison evaluation between the shared memory approach and that using shuffle instructions. The shuffle operations, along with a novel strategy for computing the ASM are presented in Sect. 4. More precisely, this section presents the \( w \)-SCAN, a parallel algorithm for computing the ASM on the GPU. Experimental results are shown in Sect. 5 and Sect. 6 concludes this work.

2. Approximate String Matching

This section defines the Approximate String Matching (ASM) problem. We begin by defining the Edit Distance (ED) of two strings [13]. Suppose that a source string \( X = x_1x_2\cdots x_m \) of length \( m \) and a destination string \( Y = y_1y_2\cdots y_n \) of length \( n \) are given. Without loss of generality, we can assume that \( m \leq n \). Suppose that we want to change \( X \) into \( Y \) using the following three operations:

- insertion of a character;
- deletion of a character;
- replacement of a character.

As example, consider the string \( X = ababa \) and string \( Y = aaabbb \). String \( X \) can be changed into \( Y \) by applying the delete (D) and insert (I) operations in sequence, as follows:

\[ D(ababa) \rightarrow D(aaba) \rightarrow I(aaa) \rightarrow I(aaab) \rightarrow I(aaabb) \rightarrow aaabbb. \]

Hence, \( X \) can be changed into \( Y \) with two delete and three insert operations. Another alternative is to use the replacement (R) operation as follows:

\[ R(ababa) \rightarrow R(aaaba) \rightarrow I(aaabb) \rightarrow aaabbb. \]

This later alternative allows to change \( X \) into \( Y \) in three operations.

The ED of two strings is the minimum number of operations to change one string into the other. For later reference, let \( ED(X, Y) \) denote the ED of \( X \) and \( Y \). The Approximate String Matching (ASM) problem is a more flexible version of the edit distance, whose task is to compute:

\[ \text{ASM}(X, Y) = \text{min}[ED(X, Y') \mid Y' \text{ is a substring of } Y]. \]

The ASM(\( X, Y \)) is small if \( Y \) has a substring similar to \( X \). Also, ASM(\( X, Y \)) \( \leq n \). A matrix \( d \) of size \((m + 1) \times (n + 1)\) can be used when computing the ASM. Each \( d[i][j] \) for \( 0 \leq i \leq m, 0 \leq j \leq n \) stores the following value:

\[ \min_{1 \leq i' \leq j} \text{ED}(x_1x_2\cdots x_i, y_{j'}y_{j'+1}\cdots y_j). \]

Note that \( x_1x_2\cdots x_i \) is a NULL string (i.e., a string with length 0) if \( i = 0 \). Once all the \( d \) values are obtained, we can compute the value of ASM(\( X, Y \)) as follows:

\[ \text{ASM}(X, Y) = \min_{0 \leq i \leq n} d[m][j]. \tag{1} \]

All values of matrix \( d \) and ASM(\( X, Y \)) can be computed by the parallel algorithm depicted in Fig. 1, which follows the description in [14]. The key idea is to compute the values of the matrix \( d \) from the top-left corner to the bottom-right corner as illustrated in Fig. 2. Let \( \text{‘}x_i \neq y_j\text{’} \) denote the binary value such that it is 1 if \( x_i \neq y_j \) and 0 if \( x_i = y_j \).
The figure shows the \( d[i,j] (0 \leq i \leq m, 0 \leq j \leq n) \) values for \( X = ababa \) and \( Y = aaabbbaa \). The final result, ASM(\( X, Y \)) = 1, is depicted in bold square in the last row.

As depicted in Fig. 1, the third for-loop of the parallel ASM controls the diagonal \( k \) of the matrix \( d \) being computed. Note that, when computing the values of \( d \), for a given \( k \), only the diagonals for \( k-1 \) and \( k-2 \) are used. As an example, suppose that \( k = 7 \) is being processed. Then, by using the values in diagonal \( k = 6 \) and \( k = 5 \), the values of \( d \) for \( k = 7 \) and \( i = 1, 2, \ldots, m \) can be computed as shown in Fig. 2. Clearly, a matrix \( e \) of size \( 3 \times (m+1) \) suffices to store the \( d \) values for \( k-2 \), \( k-1 \) and \( k \). Man et al. [7] used the above strategy, where the values of \( d[i,j] \), \( (0 \leq i \leq m, 0 \leq j \leq n) \) are stored in a matrix \( e[j \ mod \ 3][i] \) in the shared memory. It has been shown in [7] that, for a fixed \( k \), the task of computing the \( d \) values in matrix \( e \) requires \( 3m \)-read and \( m \)-write operations to the shared memory. Although the use of matrix \( e \) allows for conflict-free memory access on the DMM [12], it is still a costly solution in terms of shared memory access. Overall, the protocol in [7] requires \( (3m \times (n+m-1)) \)-read and \( (m \times (n+m-1)) \)-write operations to the shared memory for computing all the \( d \) values. In this work we take a different approach for computing \( d \) without storing \( k-2, k-1 \) and \( k \) in matrix \( e \). The proposed algorithm, termed \( w \)-SCAN, is presented in the subsequent section.

### 3. Thread Data Sharing in GPUs

Parallel programs implemented in GPUs often require the exchange of data among threads. Such communication, however, involves transferring data from a source thread to the shared memory and then to a destination thread. Clearly, such process requires the execution of at least three instructions, shared memory write, synchronize, and read. Until recently, programmers had no other choice to perform data exchange among threads. Latest GPUs, with computing capability 3.0 and higher, have changed this scenario by providing mechanisms for threads to share data without involving the shared memory, as long as these threads are confined to the same warp [10]. Recall that a warp is a group of threads (32 in current GPUs), which is the minimum size of the data processed in a SIMD CUDA multiprocessor. Inter-thread communication allows threads within the same warp to access data stored in the registers of other threads by means of shuffle instructions (a.k.a. shfl). Shuffle instructions provide a faster alternative to the conventional way of using...
the shared memory as it requires the execution of a single instruction. Furthermore, data sharing among threads using shuffle operations access data stored in local registers, which has a lower latency as compared to shared memory access.

The threads within a warp are referred to as *lanes*, with index in the range \([0, w – 1]\), where \(w\) is the warp size. As mentioned above, latest GPUs support warp shuffle operations, which are accessible to programmers via CUDA. These operations allow the exchange to occur simultaneously for all active threads within a warp. More precisely, it allows moving 4 bytes of data per thread. Four source-lane addressing modes are provided: (i) \(_\text{shfl}()\): allows a direct copy from indexed lane; (ii) \(_\text{shfl}_\text{up}()\): allows to copy data from a lane with lower ID relative to caller; (iii) \(_\text{shfl}_\text{down}()\): copies data from a lane with higher ID relative to caller; and (iv) \(_\text{shfl}_\text{xor}()\): copies data from a lane based on bitwise XOR of the caller lane ID.

Of particular interest in this work is the \(_\text{shfl}_\text{up}()\) function. This function takes two parameters: a local register variable \(var\) and the lane index \(\text{laneID}\). As an example, consider the following function call \(_\text{shfl}_\text{up}(var, 1)\). The \(_\text{shfl}_\text{up}(var, 1)\) allows to transfer the data stored in the local register variable \(var\) from a thread whose ID is immediately lower than that of the calling thread. Figure 3 shows an example of the \(_\text{shfl}_\text{up}()\) function. In Fig. 3(a), four threads \(T[0], T[1], T[2], T[3]\) use the \(_\text{shfl}_\text{up}(A)\) operation to transfer the contents in the local register \(A\) from thread \(T[i – 1]\) to \(T[i]\), \((1 \leq i \leq w – 1)\). Recall that this operation requires no shared memory access and the variable \(A\) is transferred between the threads in a warp. Also, the thread with the lowest ID (\(T[0]\)) is not affected by the \(_\text{shfl}_\text{up}()\) operation. Figure 3 (b) shows the same procedure using the shared memory to transfer data from one thread to another. In this latter case, \(2 \times (w – 1)\) memory accesses to the shared memory are required to store (\(\text{write}(A)\)) and retrieve (\(\text{read}(A)\)) to/from the shared memory.

The next subsection presents a comparison of the data movement between threads using the conventional shared memory approach and by using shuffle operations.

### 3.1 Shuffle Instruction Evaluation

This section presents an evaluation of intra-warp data sharing using both shuffle operations, which do not resort to the shared memory, and a traditional approach, which relies on the shared memory to realize the data transfer. The evaluation consists on the execution of consecutive data shuffles, similar to those shown in Fig. 3. More precisely, the experiment consists of a number of data transfer between threads within a warp. However, the data stored in thread \(T[w – 1]\) is fed back to thread \(T[0]\). This has the effect of creating a cyclic-data movement within a warp.

The above procedure has been implemented on the GTX 660 GPU, which complies with the compute capability 3.0. The GTX 660 has 4 Streaming Multiprocessors (SMp), allowing a maximum of 64 active warps per SMp. In the experiments, the number of threads have been fixed to 1024 for a varying number of CUDA blocks \(D\). In the experiments, \(D\) varies from \(2^0\) to \(2^{11}\). Table 2 shows the performance evaluation results for 20,000 consecutive intra-warp data movement, averaged over 20 runs. Recall that shuffle operations requires a single instruction to share data between threads while the shared memory requires three operations. Intuitively, one could expect that shuffle operations to reduce the computation time by two-thirds. However, as can be observed in the table, for \(D = 1\), shuffle operations reduced the computation time by at least 40 times as compared to SM operations. The reason behind this difference is mainly due to the access delay of the shared memory, which has a higher latency than that of local registers. Note that, for \(D > 8\), the number of active threads per SMp surpasses the maxi-

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Comparison evaluation of consecutive data transfers within a warp using shuffle instructions and SM instructions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CUDA Blocks</td>
<td>shfl</td>
</tr>
<tr>
<td>1</td>
<td>0.061</td>
</tr>
<tr>
<td>2</td>
<td>0.060</td>
</tr>
<tr>
<td>4</td>
<td>0.060</td>
</tr>
<tr>
<td>8</td>
<td>0.072</td>
</tr>
<tr>
<td>16</td>
<td>0.129</td>
</tr>
<tr>
<td>32</td>
<td>0.243</td>
</tr>
<tr>
<td>64</td>
<td>0.467</td>
</tr>
<tr>
<td>128</td>
<td>0.924</td>
</tr>
<tr>
<td>256</td>
<td>1.832</td>
</tr>
<tr>
<td>512</td>
<td>3.654</td>
</tr>
<tr>
<td>1024</td>
<td>7.283</td>
</tr>
<tr>
<td>2048</td>
<td>14.557</td>
</tr>
</tbody>
</table>
mum allowed number of active threads. Indeed, for \( D = 8 \), the GPU reaches its maximum occupancy of 2048 active threads per SMp. Hence, increasing \( D \) beyond 8 increases the computing time. Indeed, one can observe that doubling the number of CUDA blocks, and consequently the number of threads, increases the computing time by the same factor. This behavior is independent of whether the shuffle or SM instructions are used. Nevertheless, even in such case, the shuffle operations are still much faster than the SM operations. In fact, with \( D > 128 \), shuffle operations reduced the computation time by at least 62 times as compared to SM operations.

The \( w \)-SCAN algorithm relies on \texttt{shflup()} function to accelerate the task of computing the \( d \) values. In the rest of the paper, \texttt{laneID} = 1 and is omitted. The details of the \( w \)-SCAN algorithm are presented next.

4. Parallel ASM Algorithm

This section presents a memory-access-efficient implementation for computing the ASM on GPUs. The proposed algorithm, termed \( w \)-SCAN, significantly reduces the number of shared memory access while computing the values the intermediate values in matrix \( d \). To achieve this, \( w \)-SCAN relies on warp shuffle operations, which become available on GPUs with compute capability 3.0 and above \([10]\). The details of the \( w \)-SCAN are presented in the next subsection.

4.1 \( w \)-SCAN Algorithm

In the proposed \( w \)-SCAN algorithm, the matrix \( d \) is partitioned into \( \frac{m}{w} \) strips with \( w \) rows each. The value of \( d \) in each strip is computed in turn from the top-most strip. Each thread is assigned to a row of \( d \) in the strip and computes the values of the assigned row from the left to the right. Each thread uses three registers \( A, B \) and \( C \) to store the values of \( d \). Also, the resulting \( d \) value, in the bottom row of a strip is written to array \( f \), which is stored in the shared memory. Note that, we do not use array \( d \) to store the values of \( d \). They are stored in the registers. Figure 4 illustrates the \( w \)-SCAN algorithm for \( w = 4 \). The figure shows two-strips and the array \( f \), which is used for transferring values between the upper and the lower strip.

The marked squares correspond to the location at which \( d \) is being computed. The figure also shows the registers used when computing \( d \). The resulting \( d \) value is then stored in register \( A \).

The details of the algorithm \( w \)-SCAN are shown in Fig. 5. For each strip of \( w \) rows, \( w \)-SCAN computes the \( d \) value of the strip. The strip starts from the \texttt{top_row}-th row and spans \( w \) rows. Since we use \( w \) threads, the threads IDs take a value from 0 to \( w - 1 \). Function call \texttt{shflup(A)} returns the value of register \( A \) of the thread with \( ID - 1 \). Note that the registers \( C, B \) and \( A \) allows to retrieve the values in \( k - 2 \) and \( k - 1 \) without the need of array \( e \) to store these values in the shared memory. Instead, the thread with
Algorithm \( w\text{-SCAN} \)

\[
\begin{align*}
1 & \quad i \leftarrow ID + \text{top\_row}; \\
2 & \quad A \leftarrow i; \\
3 & \quad B \leftarrow i - 1; \\
4 & \quad X \leftarrow X[i]; \\
5 & \quad \text{for } k \leftarrow [1, m+n-1] \text{ do in parallel} \\
6 & \quad \quad \begin{align*}
7 & \quad j \leftarrow k - ID; \\
8 & \quad \quad \text{if } (1 \leq j < ID) \\
9 & \quad \quad \quad C \leftarrow B; \\
10 & \quad \quad \quad \text{//top\_row of the strip} \\
11 & \quad \quad \quad \text{if } ((ID \mod w) = 0) \text{ then} \\
12 & \quad \quad \quad \quad B \leftarrow f[ID/w]; \\
13 & \quad \quad \quad \text{else} \\
14 & \quad \quad \quad \quad B \leftarrow \text{shfl\_up}(A); \\
15 & \quad \quad \quad A \leftarrow \min(A + 1, B + 1, C + (X \not= y_j)); \\
16 & \quad \quad \quad \text{//bottom\_row of the strip} \\
17 & \quad \quad \quad \text{if } (ID \mod w = (w - 1)) \text{ then} \\
18 & \quad \quad \quad \quad f[ID/w + 1] \leftarrow A; \\
19 & \quad \quad \end{align*}
20 & \end{align*}
\]

Fig. 5  Pseudo-code of the \( w\text{-SCAN} \) algorithm.

The lowest (bottom\_row) and highest (top\_row) IDs in a strip access the array \( f \) to exchange information between the threads located at different strips. That is, the array \( f \) holds all the values being transferred from the strip \( s_j \) to strip \( s_k \) \((0 \leq j < k \leq \frac{m}{w} - 1)\). By shuffling the transferred results along \( k \), the remaining threads within a strip can compute the \( d \) values based on local register information. Hence, by using three registers, the threads in a strip can access the values in \( k-1 \) and \( k-2 \) to compute \( d \). Once all values of \( d \) are computed, the bottom-most thread in strip \( \frac{w}{w} - 1 \) holds the final ASM(\( X, Y \)) value.

One can confirm that the \( w\text{-SCAN} \) algorithm provides a conflict-free memory access to array \( f \). For a fixed \( k \), \( w\text{-SCAN} \) requires \( \left(\frac{w}{w}\right)\)-write and \( \left(\frac{w}{w}\right)\)-read operations to array \( f \). Recall from Sect. 2 that the O-ASM algorithm in [7] requires \( 3m\)-read and \( m\)-write shared memory accesses to complete the same task. The for\_loop in line 5 iterates for \( m+n-1 \) times. Hence, overall, \( w\text{-SCAN} \) algorithm requires \( \left(\frac{w}{w}\right) \times (m+n-1)\)-write and \( \left(\frac{w}{w}\right) \times (m+n-1)\)-read operations to the shared memory for transferring values among the \( \left(\frac{w}{w}\right) \) strips while computing the \( d \) values.

The process of transferring the values in arrays \( Y \) and \( X \) from the global memory and writing the results back is similar to the O-ASM presented by [7]. More precisely, the source string \( X = x_1x_2 \cdots x_m \) of length \( m \) and the destination string \( Y = y_1y_2 \cdots y_n \) of length \( n \) are transferred from the host and stored to the global memory of the device. In O-ASM, when the kernel starts, both strings are transferred to the shared memory. In \( w\text{-SCAN} \), however, the string \( X \) is read from the global memory by the active threads, where each thread stores an element \( x_i \) \((1 \leq i \leq m)\) on its local register. Hence, in comparison to O-ASM, \( w\text{-SCAN} \) reduces the time to transfer the source string \( X \) to the shared memory. As in [7], string \( Y \) is partitioned into \( D \) substrings \( Y = Y_0, Y_1, \ldots, Y_{D-1} \). The partitioning is performed in such a way that each substring of size \( 2m \) is included in one of the \( Y_i \)’s substrings. Then, the ASM of \( X \) is computed in parallel for each substring \( Y_i \). In comparison to O-ASM, \( w\text{-SCAN} \) reduces the number of shared memory accesses by resorting to shuffle instructions. In \( w\text{-SCAN} \), a single read and write to array \( f \) suffices to transfer values among different strips. This strategy allows \( w\text{-SCAN} \) to reduce the number of shared memory accesses necessary to compute the \( d \) values. For each substring \( Y_i \), the bottom-most thread is responsible for storing the resulting value of \( d \), which is then written to the global memory. The final result is computed by taking the minimum value among the \( Y_i \)’s substrings such that \( \text{ASM}(X, Y) = \min(\text{ASM}(X, Y_i)) \text{for } (0 \leq i \leq D - 1) \).

The next subsection presents the experimental results for the \( w\text{-SCAN} \) algorithm.

5. Experimental Results

We have implemented the \( w\text{-SCAN} \) algorithm for the HMM on the GeForce GTX 980 GPU [15]. The GTX 980 has 16 streaming multiprocessors, warp size (\( w \)) of 32 and 4GB of memory. The proposed \( w\text{-SCAN} \) is compared to the O-ASM, a parallel algorithm on GPU for computing the ASM proposed in [7]. The \( w\text{-SCAN} \) uses the strategy in [7] for handling the arrays \( X \) and \( Y \). In the experiments, the nvcc compiler version 6.5.13, CUDA version 6.5 and CUDA driver version 6.5 were used. The results for a sequential ASM on a single CPU using the GNU Compiler Collection (GCC) version 5.3 is also presented. The sequential algorithm was executed on an Intel i7 4790 CPU at 2.9GHz.

The input \( Y \) is partitioned into \( D = 128, 256, 512, 1024, 2048 \) substrings and \( D \) CUDA blocks of \( m \) threads are invoked to compute \( \text{ASM}(X, Y_i)(0 \leq i \leq D - 1) \). As in [7], the characters of \( X \) and \( Y \) are treated as 8-bit unsigned char. Since \( X \) and \( Y \) are random 0/1 strings, “\( x_i \neq y_j \)” is true with probability \( \frac{1}{2} \). Such strings are unfavorable for GPUs, because the resulting values of “\( x_i \neq y_j \)” by all threads in a warp are not the same with high probability.

Once the strings \( X \) and \( Y \) are initialized on the host, they are transferred to the GPU global memory using the CUDA cudaMemcpy() function, as shown in Fig. 6. The GPU computing time refers to the kernel execution time, excluding the host-to-device transfer time. The GPU runs the \( w\text{-SCAN} \) algorithm and computes the \( \text{ASM}(X, Y_i)(0 \leq i \leq D - 1) \) using \( D \) CUDA blocks in parallel. The computing time stops right before the \( \text{ASM}(X, Y_i) \) values are transferred to the CPU where the final ASM result is computed.

Table 3 shows the computing time for the \( w\text{-SCAN} \) and O-ASM and their respective speed-up over the CPU. The table shows the running time in milliseconds for \( Y \) with \( 4M \text{ characters} \) and \( X \) with 32, 64, 128, 256, 512, 1024 characters. Note that the size of \( X \) is restricted the available
Table 3  Computing time (ms) for the w-SCAN and the O-ASM [7] with a varying number of GPU CUDA blocks $D$ on the GTX 980 GPU with parameter $|Y| = 2^{22}$.

| $|X|$ | Algorithm | $D = 128$ | $D = 256$ | $D = 512$ | $D = 1024$ | $D = 2048$ | CPU Time | Speed-up |
|------|-----------|------------|------------|------------|------------|------------|----------|----------|
| 32   | w-SCAN    | 10.37      | 5.34       | 3.07       | 2.95       | 2.95       | 592.18   | 200.96   |
|      | O-ASM     | 33.81      | 17.07      | 7.61       | 5.81       | 4.91       | 1148.31  | 120.55   |
| 64   | w-SCAN    | 11.02      | 5.73       | 4.28 5.38  | 4.38       | 4.55       | 2308.06  | 268.57   |
|      | O-ASM     | 34.93      | 19.02      | 12.95      | 10.27      | 10.09      | 1428.85  | 113.85   |
| 128  | w-SCAN    | 11.76      | 8.18       | 8.75       | 8.73       | 9.77       | 282.27   | 219.77   |
|      | O-ASM     | 43.95      | 26.00      | 21.02      | 20.07      | 21.48      | 4255.89  | 114.97   |
| 256  | w-SCAN    | 16.79      | 17.39      | 18.39      | 19.71      | 22.44      | 4528.69  | 269.77   |
|      | O-ASM     | 56.42      | 42.16      | 41.14      | 44.04      | 49.30      | 110.07   | 90.95    |
| 512  | w-SCAN    | 36.27      | 37.74      | 40.37      | 44.76      | 55.47      | 9055.22  | 249.66   |
|      | O-ASM     | 95.17      | 87.71 85.46 | 93.61      | 114.39     | 149.25     | 105.95   | 97.35    |
| 1024 | w-SCAN    | 91.96      | 87.25      | 88.37      | 108.65     | 151.66     | 3115.65  | 205.37   |
|      | O-ASM     | 186.04     | 184.07     | 220.87     | 268.60     | 342.29     | 17918.65 | 97.35    |

Table 4  Number of executed write instructions (shown in million instructions).

| $|X|$ | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN |
|------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|
| 32   | 4.34  | 4.21   | 4.33  | 4.23   | 4.38  | 4.26   | 4.43  | 4.33   | 4.33  | 4.46   | 4.33  | 4.46   |
| 64   | 8.56  | 8.44   | 8.60  | 8.50   | 8.69  | 8.60   | 8.69  | 8.60   | 8.81  | 9.19   | 9.24   |
| 128  | 17.06 | 16.97  | 17.21 | 17.15  | 17.51 | 17.53  | 18.10 | 18.28  | 19.30 | 19.79  |
| 256  | 34.24 | 34.26  | 34.81 | 34.96  | 35.93 | 36.37  | 38.17 | 39.19  | 42.66 | 44.83  |
| 512  | 69.40 | 69.81  | 71.57 | 72.54  | 75.92 | 77.98  | 84.61 | 88.87  | 101.97| 110.62 |
| 1024 | 142.82| 144.84 | 151.36| 155.53 | 168.44| 176.90 | 202.60| 219.64 | 270.93| 305.14 |

Table 5  Number of executed read instructions (shown in million instructions).

| $|X|$ | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN | O-ASM | w-SCAN |
|------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|-------|--------|
| 64   | 42.14 | 8.44   | 42.33 | 8.49   | 42.72 | 8.60   | 43.50 | 8.81   | 45.06 | 9.23   |
| 128  | 84.60 | 16.96  | 85.32 | 17.15  | 86.75 | 17.53  | 89.62 | 18.28  | 95.35 | 19.78  |
| 256  | 170.50| 34.25  | 173.25| 34.96  | 178.75| 36.36  | 189.73| 39.17  | 211.68| 44.79  |
| 512  | 346.20| 69.81  | 356.95| 72.53  | 378.43| 77.96  | 421.40| 88.83  | 507.25| 110.56 |
| 1024 | 713.22| 144.83 | 755.67| 155.51 | 840.38| 176.87 | 1010.40| 219.58 | 1350.04| 305.00 |

Fig. 6  Data transfer between the host (CPU) and the device (GPU board).
nvprof, the values in Table 4 are similar for both w-SCAN and O-ASM, even though w-SCAN uses a single thread per strip to write intermediate values in array \( f \). In contrast, the O-ASM uses all the \( w \) threads to write the values in matrix \( e \). As the number of strips in w-SCAN range from 1 to 32, with \( w = 32 \), the shared memory access improvements provided by w-SCAN are not explicit in the table. Table 5 shows the number of read instructions for w-SCAN and O-ASM. As expected, w-SCAN presents comparable results in terms of the number of read and write instructions. The O-ASM, however, requires three read operations to compute the values in matrix \( e \) plus one read to fetch the values from string \( Y \). The nvprof results show that O-ASM demands for over 4 times the number of read operations used by w-SCAN, which is consistent to the numerical results.

6. Conclusion

This work presented a parallel algorithm for computing the approximate string matching on the GPU. The proposed algorithm, termed w-SCAN, explores the use of shuffle operations, available on recent NVIDIA GPUs, to speed-up the ASM computation. We showed that the w-SCAN is able to reduce the amount read/write operations while computing intermediate values of the ASM. Experimental results show that, while computing the ASM, the proposed w-SCAN implementation provides speed-up exceeding two fold as compared to previous results. The above results confirm that w-SCAN provides a fast alternative for computing the approximate string matching using recent GPUs.

References


Lucas Saad Nogueira Nunes received B.E. degree in Computer Engineering in 2015 from University of Brasilia. Mr. Nunes is currently enrolled in the Master Course in Informatics at the University of Brasilia where he is pursuing his Masters Degree. His interest includes reconfigurable architectures, parallel computing, and algorithms and architectures.

Jacir Luiz Bordim received B.Sc. and M.Sc. degrees in Computer Science in 1994 and 2000, respectively. Received the Ph.D. degree in Information Science from Japan Advanced Institute Of Science And Technology in 2003, with honors. He worked as a researcher at ATR-Japan from 2003 to 2005. Since 2005 he is an Associate Professor with the Department of Computer Science at University of Brasilia. Dr. Bordim has published and served in many international conferences and journals. His interest includes mobile computing, collaborative computing, trust computing, distributed systems, opportunistic spectrum allocation, MAC, routing protocols and reconfigurable computing.
Yasuaki Ito received B.E. degree from Nagoya Institute of Technology (Japan), M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and D.E. degree from Hiroshima University (Japan), in 2010. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Associate Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.

Koji Nakano received the BE, ME and Ph.D degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992-1995, he was a Research Scientist at Advanced Research Laboratory, Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. His research interests include image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.