Massively Parallel Computations of the LZ-complexity of Strings

Alexander Belousov
Electrical and Electronics Engineering Department
Ariel University
Ariel, Israel
alex.blsv@gmail.com

Joel Ratsaby
Electrical and Electronics Engineering Department
Ariel University
Ariel, Israel
ratsaby@ariel.ac.il

Abstract — **We introduce a new parallel algorithm LZMP for computing the Lempel-Ziv complexity of a string of characters from a finite alphabet. The LZ-complexity is a mathematical quantity that is related to the amount of non-redundant information that a string contains. It has been recently shown to be useful in pattern recognition in measuring the distance between various kinds of input patterns [1],[2].**

We implement the algorithm on a software parallel-computing platform (CUDA) which works with a Graphical Processing Unit (GPU) on a TESLA K20c board. The GPU has 2,496 computing cores and 5G bytes of global memory. This platform enables high increase in computing performance.

We exploit multiple places in the sequential version of the code and use CUDA to implement these code sections to run in parallel on the GPU cores. For example, the string XYZMXZXYZKR has a complexity of 7. The sequential version of the algorithm takes 27 time steps to compute this while the parallel version takes only 11 time steps. The advantage of the parallel implementation over the sequential one becomes more relevant as the string's length increases.

Our results show that for single strings of length up to 5k bytes there is no significant improvement but for strings of length greater than 5k the speedup factor grows at a linear rate with respect to the string length. When we compute the LZ-complexity of multiple strings all of length 48k bytes in a parallel our results indicate that the speedup ratio is approximately 100. This is advantageous for pattern recognition where the object to be recognized can be represented as a collection of substrings and LZ-complexity based distance ca be computed to each of them in parallel. We provide our CUDA source code [7].

I. Introduction

The Lempel-Ziv complexity [3] of a finite string S is proportional to the minimal number of substrings that are necessary to produce *S* via a simple copy operation. For instance, the string *S* = *ababcabcabcbaa* can be constructed from the five substrings, *a,b,abc,abcabcb,aa*, and therefore its LZ-complexity equals 5. This complexity measure inspired the well-known universal compression algorithms of Lempel Ziv [5,6]. It has recently been used in pattern recognition and classification through the introduction of a new string distance function [4]. This distance function has inspired a large number of applications, especially in the field of biological-sequence analysis. In [1,2] a new universal image distance (UID) was introduced which is based on the LZ-complexity. The sequential algorithm for computing the LZ-complexity is $O(n^2)$, where *n* is the length of the string. This means that computing the UID between typical-size images is impractical for most real-time applications. Hence in this paper we introduce a parallel algorithm for computing the LZ-complexity which has a linear speedup relative to the sequential algorithm, with respect to the number of cores. This makes computing the UID practical.

II. LZ-complexity

The definition of LZ-complexity follows [3]: let *S*,*Q* and *R* be strings of characters that are defined over the alphabet *A*. Denote by $l(S)$ the length of *S*, and $S(i)$ denotes the element of *S*. We denote by $S(i, j)$ the sub string of *S* which consists of characters of *S* between position *i* and *j* (inclusive). An extension $R = SQ$ of *S* is reproducible from *S* (denoted as $S \to R$) if there exists an integer $p \le l(S)$ such that $Q(k) = R(p+k-1)$ for $k = 1, \dots, l(Q)$.

For example, $aacgt \to aacgtcgtcg$ with $p = 3$ and $aacgt \to aacgtac$ with $p = 2$. *R* is obtained from *S* (the seed) by first copying all of *S* and then copying in a sequential manner $l(Q)$ elements starting at the p^{th} location of *S* in order to obtain the *Q* part of *R*.

A string *S* is *producible* from its prefix $S(1, j)$ (denoted $S(1, j) \Rightarrow R$), if $S(1, j) \to S(1, l(S) - 1)$.

For example, $aacgt \Rightarrow aacgtac$ and $aacgt \Rightarrow aacgtacc$ both with pointers $p = 2$. The production adds an extra 'different' character at the end of the copying process which is not permitted in a reproduction.

Any string *S* can be built using a *production process* where at its i^{th} step we have the production $S(1, h_{i-1}) \Rightarrow S(1, h_i)$ where h_i is the location of a character at the i^{th} step. (Note that $S(1, 0) \Rightarrow S(1, 1)$).

An *m*-step production process of *S* results in parsing of *S* in which $H(S) = S(1, h_1) \cdot S(h_1+1, h_2) \cdots S(h_{m-1}+1, h_m)$ is called the *history* of *S* and $H_i(S)$ $(S) = S(h_{i-1}+1, h_i)$ is called the i^{th} component of $H(S)$. For example for $S = aacgtacc$ we have $H(S) = a \cdot ac \cdot g \cdot t \cdot acc$ as the history of

S. If *S*(1, h_i) is not reproducible from *S*(1, h_{i-1}), then component $H_i(S)$ is called *exhaustive* meaning that the copying process cannot be continued and the component should be halted with a single character *innovation*. A history is called exhaustive if each of its components (except maybe the last one) is exhaustive. Every string *S* has an unique exhaustive history [3]. Let us denote by $c_H(S)$ the number of components in a history of *S*. The LZ complexity of *S* is $c(S)=min\{c_H(S)\}$ where the minimum is over all histories of *S*. It can be shown that $c(S)$ = $c_E(S)$ where $c_E(S)$ is the number of components in the exhaustive history of *S*.

III. GPU-based parallel processing

Graphical processing unit (GPU) is a multi-processor electronic chip specialized for matrix and vector operations to do geometrical operations in 3D graphics. GPUs have been recently gaining popularity also for non-graphical processing due to their highly parallel computing architecture. This architecture typically consists of thousands of cores that operate concurrently on several gigabytes of global memory. There are several software platforms that allow provide API for main languages such as C and C++ and enable programmers to produce code that runs on a GPU. We use the CUDA™ platform is freely provided by the nVIDIA corporation. We use a TESLA K20c GPU which delivers 3.5 Tflops. Compared to our AMD 6-core Phenom II CPU which delivers only several Gflops.

IV. Algorithms

We start by presenting the pseudo-code of the sequential LZ-complexity algorithm and then introduce the parallel algorithm, which we call Lempel-Ziv-Massively-Parallel (LZMP) algorithm. Let us denote by $T_s(n)$ the time that it takes for the sequential algorithm to compute the LZ-compelxity of a string of length *n,* in the worst case. Let $T(n,p)$ be the time it takes for the parallel algorithm, running on *p* cores, to compute the LZ-complexity of the string of length *n,* in the worst case.

Define by $S_p:=T_s(n)/T(n,p)$ the speedup factor achieved by the parallel algorithm. Algorithm LZMP introduced below has a linear speedup factor with respect to *p*, that is, $S_p=p$. We now present the pseudo code of the sequential algorithm and then the pseudo code of the parallel algorithm (LZMP). All variables in small caps are integer-valued. We use large-capital letters for representing sets, collection, and memory buffers.

A. Pseudo code of the sequential algorithm

```
1.   Input: $S$ string of $n$ elements
         $S=\langle s_1, s_2, s_3, \ldots, s_n\rangle$
     Initialize
2.   $H$ history buffer
3.   $m$ history buffer length, Initialize $m:=0$
4.   $max$ variable for storing a greater counter of steps
5.   $d$ the number of components in the exhaustive history
     Initialize $d$ , $d:=0$
     // Scan $S$ string from a beginning to end, let the
     // sequence find new substrings (components) and
     // add them to history buffer $H$ .
6.   while ( $m<n$ )
     6.1. Initialize $max:=0$
     6.2. for( $l=0$ to $m$ )
          6.2.1.     initialize variable $i:=0$
          6.2.2.     initialize variable $k:=0$
// compare elements in history buffer with elements in $S$ string.
          6.2.3.     while ( $H[l+k]=S[m+i]$ )
                         $k:=k+1$
                         $i:=i+1$
                         if( $l+k=m$ or $m+i=n$ )
                             break;
                         end if;
          6.2.4.     end while;
// check if history buffer is over and current is not
          6.2.5.     if( $l+k=m$ and $m+i<n$ )
     // Continue to scan and compare
     // elements in $S$ string.
                         initialize $z:=m$
                         while( $S[z]=S[m+i]$ )
                             $z:=z+1$
                             $i:=i+1$
                             if( $m+i=n$ )
                                 break;
                             end if;
                         end while;
          6.2.6.     end if;

     // check if $i$ is greater than $max$ .
     // If greater, put $i$ value to $max$ ,
                 if( $i>max$ )
                     $max:=i$
                 end if;
     6.3. end for;
        //Append new substring to history buffer $H$
     6.4.
          $H:=H+substring(S[m], S[m+max+1])$
     6.5. $d:=d+1$
     6.6. $m:=m+max+1$
7.   end while;
8.   Output: $d$ The LZ-complexity of a $S$ string.
```

We now compute the $T_s(n)$ of the sequential algorithm. We consider the worst-case string, which is one where the reproduction loops (6.2.3 – 6.2.6) do not occur. In this case as we scan *S* from the left, for the character at position m it takes $m-1$ attempts to reproduce the substring starting at m, which takes $\sum_{m=1}^{n}(m-1)$ units of time. Therefore the total time for computing the LZ-compelxity of a string of length *n* is $(n/2)(n+1)-n=O(n^2)$.

We now introduce the parallel algorithm LZMP.

B. *Pseudo code of algorithm LZMP*

```
1.  Input: $S$ string of $n$ elements
        $S=\langle s_1, s_2, s_3, \ldots, s_n \rangle$
    Initialize
2.  $H$ history buffer
3.  $m$ history buffer length, Initialize $m:=0$
4.  $T_q$ Thread with ID = $q$
5.  $p$ total number of cores
6.  $SM$ shared memory variable for storing a greater
    counter of steps, shared memory visible for all threads.
7.  $d$ the number of components in the exhaustive
    history
    Initialize $d$ , $d:=0$
  // Scan $S$ string from a beginning to end, let the
  // algorithm find new substrings (components) and
  // add them to history buffer $H$ .
8.  launch parallel Threads $T_q$ , $0 \le q < p$
9.  while ( $m<n$ )
    9.1. Initialize $SM:=0$
    9.2. for( $l=0$ to $\lfloor m/p \rfloor$ )
  // create new index $j$ that depends from
  // Thread with ID = $q$ ( $T_q$ )
    9.2.1.   initialize variable $j=q+l \cdot p$
    9.2.2.   if( $j<m$ )
             initialize variable $i_j:=0$
             initialize variable $k_j:=j$
             initialize variable $h_j:=m-j$
  // Let each Thread scan and compare elements in
  // history buffer with elements in $S$ string.
    9.2.2.1.      while ( $H[k_j]=S[m+i_j]$ )
                    $k_j:=k_j+1$
                    $i_j:=i_j+1$
                    $h_j:=h_j-1$
                    if( $h_j=0$ or $m+i_j=n$ )
                       break;
                    end if;
    9.2.2.2.      end while;
  // Check if history buffer is over and current is not
    9.2.2.3.      if( $h_j=0$ and $m+i_j<n$ )
  // Let each Thread continue to scan and compare
  // elements in $S$ string.
                    initialize $z_j:=m$
                    while( $S[z_j]=S[m+i_j]$ )
                       $z_j:=z_j+1$
                       $i_j:=i_j+1$
                       if( $m+i_j=n$ )
                          break;
                       end if;
                    end while;
    9.2.2.4.      end if;
  // Let each Thread $T_j$ check if $i_j$ is greater than $SM$ .
  // If greater, put $i_j$ value to $SM$ ,
    9.2.2.5.      if( $i_j>SM$ )
                       $SM:=i_j$
    9.2.2.6.      end if;
    9.2.3.   end if;
    9.3. end for;
    9.4. Synchronize all Threads $T_q$ .
    9.5. if( $T_q=T_0$ )
  //Append new substring to history buffer $H$
       $H:=H+substring(S[m], S[m+SM+1])$
             $d:=d+1$
             $m:=m+SM+1$
    9.6. end if;
    9.7. Synchronize all Threads $T_q$ .
10. end while;
11. Output: $d$ The LZ-complexity of a $S$ string.
```

We now compute the $T(n, p)$ of the parallel algorithm. The worst case string is one for which the reproduction step (which is implemented via the two inner while-loops) does not occur. In this case it takes

$$\sum_{i=1}^{p} 1+\sum_{i=1+p}^{2p} 2+\sum_{i=1+2p}^{3p} 3+\ldots+\sum_{i=n-p}^{n}(n/p)$$

therefore the total time for computing the LZ-complexity of a string of length n is bounded from above by

$p\sum_{i=1}^{\lceil n/p \rceil} i=(n/2)((n/p)+1)$. Therefore the speed up is

$$\frac{T_s(n)}{T(n,p)}=\frac{(n/2)(n+1)-n}{(n/2)((n/p)+1)}=p\left(\frac{n-1}{n+p}\right)$$

which is approximately equal to p if $\lim_{n\to\infty}(p/n)=0$.

V. ANALYSIS

We now analyze the functionality and compare between the two algorithms presented in the previous section by considering an example string XYZMXZXYZKR. Table I shows the dry-run of the sequential algorithm. The left column represents the time units, the second column shows the progress in reading the characters of the string *S.* The bold-font character is at position *m+i* (see step 6.2.3 in section IV.A).

The third column shows the content of the history buffer. The bold-font character is at position $l+k$ (step 6.2.3). The sixth column displays the process of finding the longest substring (called candidate) that starts at m^{th} position of *S* and ends at $(m+max)^{th}$ character of *S*. Each candidate is produced by a copying process, the l^{th} process starts at the $(l+k)^{th}$ character of the history buffer. The column contains the list of candidates that have been seen so far separated by a comma. When there are no more candidates to be considered, we add to the dictionary the one whose length is maximal. The eighth column shows the value of *max*. The last column displays the current value of the LZ-complexity of *S* which is the number of components added to the dictionary up to the current time. At time 27 the computation halts and the LZ-complexity value of *S* is 7.

TABLE I. SEQUENTIAL COMPUTATION

t (time)	String S	History Buffer	m+i	k	Temp candidate	component added to Dictionary	max	LZ value
1	**X**YZMXZXYZKR	--	0	0	X	X	1	**1**
2	X**Y**ZMXZXYZKR	**X**	1	0	Y	Y	1	**2**
3	XY**Z**MXZXYZKR	**X**Y	2	0	--	--	0	2
4	XY**Z**MXZXYZKR	X**Y**	2	1	Z	Z	1	3
5	XYZ**M**XZXYZKR	**X**YZ	3	0	--	--	0	3
6	XYZ**M**XZXYZKR	X**Y**Z	3	1	--	--	0	3
7	XYZ**M**XZXYZKR	XY**Z**	3	2	M	M	1	**4**
8	XYZM**X**ZXYZKR	**X**YZM	4	0	--	--	1	4
9	XYZMX**Z**XYZKR	X**Y**ZM	5	1	XZ	--	2	4
10	XYZM**X**ZXYZKR	XY**Z**M	4	2	XZ	--	2	4
11	XYZM**X**ZXYZKR	XYZ**M**	4	3	XZ	XZ	2	**5**
12	XYZMXZ**X**YZKR	**X**YZMXZ	6	0	--	--	0	5
13	XYZMXZX**Y**ZKR	X**Y**ZMXZ	7	1	--	--	0	5
14	XYZMXZXY**Z**KR	XY**Z**MXZ	8	2	--	--	0	5
15	XYZMXZXYZ**K**R	XYZ**M**XZ	9	3	XYZK	--	4	5
16	XYZMXZ**X**YZKR	XYZM**X**Z	6	4	XYZK,--	--	4	5
17	XYZMXZX**Y**ZKR	XYZMX**Z**	7	5	XYZK, XY	XYZK	4	**6**
18	XYZMXZXYZK**R**	**X**YZMXZXYZK	10	0	--	--	0	6
19	XYZMXZXYZK**R**	X**Y**ZMXZXYZK	10	1	--	--	0	6
20	XYZMXZXYZK**R**	XY**Z**MXZXYZK	10	2	--	--	0	6
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
27	XYZMXZXYZK**R**	XYZMXZXYZ**K**	10	9	R	R	1	7

TABLE II. LZMP COMPUTATION

t	String S	History Buffer	thread0 m+i k, temp			thread1 m+i k, temp			thread2 m+i k, temp			thread3 m+i k, temp			thread4 m+i k, temp			thread5 m+i k, temp			thread6 m+i k, temp			...	Comp. added to Dict.	max	LZ value
1	**X**YZMXZXYZKR	--	0	0	--	Inactive			Inactive			Inactive			Inactive			Inactive			Inactive			...	X	1	**1**
2	X**Y**ZMXZXYZKR	**X**$_0$	1	0	--	Inactive			Inactive			Inactive			Inactive			Inactive			Inactive			...	Y	1	**2**
3	XY**Z**MXZXYZKR	**X**$_0$**Y**$_1$	2	0	--	2	1	--	Inactive			Inactive			Inactive			Inactive			Inactive			...	Z	1	**3**
4	XYZ**M**XZXYZKR	**X**$_0$**Y**$_1$**Z**$_2$	3	0	--	3	1	--	3	2	--	Inactive			Inactive			Inactive			Inactive			...	M	1	**4**
5	XYZM**X**ZXYZKR	**X**$_0$**Y**$_1$**Z**$_2$**M**$_3$	4	0	X	4	1	--	4	2	--	4	3	--	Inactive			Inactive			Inactive			...	--	0	4
6	XYZMX**Z**XYZKR	X **Y**$_0$Z M	5	1	XZ	Inactive			Inactive			Inactive			Inactive			Inactive			Inactive			...	XZ	2	**5**
7	XYZMXZ**X**YZKR	**X**$_0$**Y**$_1$**Z**$_2$**M**$_3$**X**$_4$**Z**$_5$	6	0	X	6	1	--	6	2	--	6	3	--	6	4	X	6	5	--	Inactive			...	--	2	5
8	XYZMXZX**Y**ZKR	X **Y**$_0$Z M X **Z**$_4$	7	1	XY	Inactive			Inactive			Inactive			7	5	XY	Inactive			Inactive			...	--	2	5
9	XYZMXZXY**Z**KR	X Y **Z**$_0$M X Z	8	2	XYZ	Inactive			Inactive			Inactive			Inactive			Inactive			Inactive			...	--	3	5
10	XYZMXZXYZ**K**R	X Y Z **M**$_0$X Z	9	3	XYZK	Inactive			Inactive			Inactive			Inactive			Inactive			Inactive			...	XYZK	4	**6**
11	XYZMXZXYZK**R**	**X**$_0$**Y**$_1$**Z**$_2$**M**$_3$**X**$_4$**Z**$_5$**X**$_6$**Y**$_7$**Z**$_8$**K**$_9$	10	0	--	10	1	--	10	2	--	10	3	--	10	4	--	10	5	--	10	6	--	...	R	1	**7**

Active threads
-- := empty
$\underline{x_i}$:= the character on which Thread i is currently working

Table II shows a dry-run of the parallel algorithm on the above example. The middle columns display the activity of the p threads that run in parallel. Each thread attempts to produce one candidate substring that starts at position m in the string S. Each thread has its own variable i which points to the current position of the substring associated with the thread. The total time needed for computing the LZ-complexity of S by LZMP is 11, compared to the total time of 27 for the sequential algorithm. As can be seen in this example, there are several inactive threads due to the fact that the history is short. For a longer string S, as the length of the history gets larger and closer to the number of threads p, more threads become active.

VI. Experiments and Results

The computing platform consists of a 2.8GHz AMD Phenom©II X6 1055T Processor with number of cores n = 6 and the operating system is Ubuntu 12.04 LTS. Note that the sequential algorithm runs on a single core of this processor, that is, our speedup results are with respect to the time that it takes the sequential algorithm to run on a single core. The GPU hardware is a Tesla K20C board with a single GK110 GPU from NVIDIA. This GPU is based on the Keppler architecture (with compute capabilities of 3.5). The CUDA is release 6.0.

We conducted two experiments, the first aims at estimating the speedup of algorithm LZMP on the task of computing the LZ-complexity of a single random string of ASCII characters of length n and varied the value of n in the range of $1\text{k} \le n \le 1000\text{k}$. The number of threads per block is 1,024 and we use a single block for the computation. In the second experiment we constructed M random character strings, where $1 \le M \le 140$, each string of length 48k. We assigned each string to a distinct block of the GPU and allocated 1,024 threads per block of the GPU. Note that the size of the shared memory on each block is 48k hence each of the strings occupies maximum shared memory on its corresponding block. Using this setup we estimated the speedup factor of computation of the LZ-complexity for all of these M strings in parallel. Figure 1 displays the result for experiment 1. We see the speedup factor as a function of the string size n.

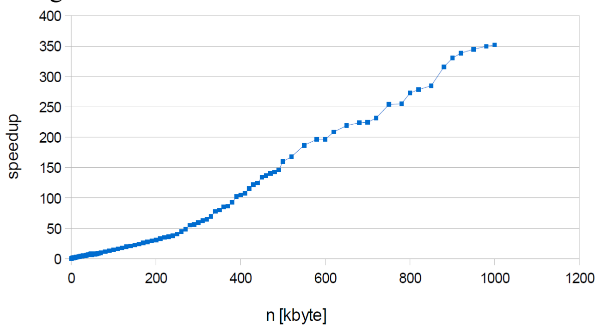

Fig. 1

Note that the speedup keeps increasing with n. In the interval of $1 \le n \le 5\text{k}$ the speedup is less than 1 hence making the parallel implementation unuseful for string lengths that are shorter than 5k. In order to appreciate the speedup, let us compare the absolute running times for $n = 1000\text{K}$: the CPU time is 38.4 hours and the GPU only 6 minutes. Figure 2 displays the result of experiment 2. As can be seen, the speedup factor increases non-linearly and converges to approximately 100. For instance, to compute the LZ-complexity for 140 strings of length 48k each, the CPU requires 17.9 minutes while it takes the GPU only 10 seconds.

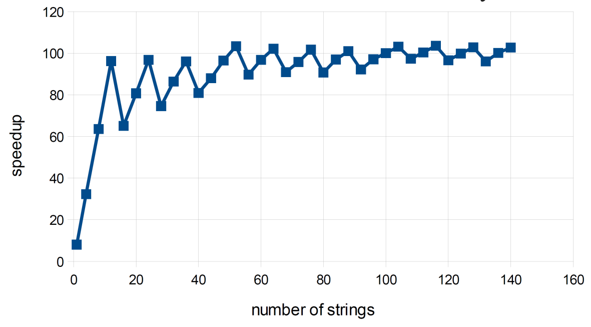

Fig. 2

VII. Conclusions

We introduced a parallel algorithm, LZMP, for computing the LZ-complexity of a string of characters. The algorithm has a theoretical speedup factor $S_p = p$ where p is the number of cores. We tested the algorithm on two problems, one of computing the LZ-complexity of a string of length n, for $1\text{k} \le n \le 1000\text{k}$ bytes and the second is of computing the LZ-complexity of many strings each of length 48k bytes. For the first problem the results indicate that there is approximately a linear speedup with respect to n, and in the second problem, we see a maximal speedup of about 100 with respect to the number of strings. With this speedup, it becomes viable to do pattern recognition that use distance-measures that are based on the LZ-complexity such as in [2,3].

VIII. Acknowledgment

We acknowledge the support of the nVIDIA corporation in providing the necessary GPU hardware.

References

[1] U. Chester, J. Ratsaby. Universal distance measure for images, *Proc. of the 27th IEEE Convention of Electrical and Electronics Engineers in Israel (IEEEI'12)*, pp. 1-4, Eilat, Israel, Nov. 14-17, 2012.

[2] U. Chester, J. Ratsaby. Image classification and clustering using a universal distance measure, in N. Brisaboa, O. Pedreira, and P. Zezula (Eds.), *Proc. of the 6th Int'l conf. on Similarity Search and Applications (SISAP 2013)*, Springer LNCS 8199, pp. 59-72, La Coruna, Spain, Oct. 2-4, 2013.

[3] A. Lempel, J. Ziv, On the Complexity of Finite Sequences, *IEEE Transactions on Information Theory,*, vol.22, no.1, pp.75-81, 1976.

[4] K. Sayood and H. H. Otu. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics*, 19(16):2122–2130, 2003.

[5] J. Ziv, A. Lempel, (1977) A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23, 337–343.

[6] J. Ziv, A. Lempel, (1978) Compression of indiviual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24, 530–536.

[7] CUDA source code of LZMP, http://www.ariel.ac.il/sites/ratsaby/Code/LZMP.zip