Ziv-Lempel compression
with the Connex Engine

Dominique Thiebaut\textsuperscript{1} and Gheorghe Ştefan\textsuperscript{2}

\textsuperscript{1}Dept. of Computer Science, Smith College, Northampton, MA
\textsuperscript{2}Dept. of Electronics, Politehnica University of Bucharest, Romania

January 21, 2002

Abstract

This paper presents an algorithm implementing the Ziv-Lempel compression algorithm using the Connex Engine circuit. The algorithm illustrates the power of the parallelism of this in-memory device that performs basic string operations such as search, insert and delete character-operations in one clock cycle. The complexity of the algorithm is $O(N)$, where $N$ is the length of the string to be compressed, and with a multiplicative constant less than 50 CPU cycles plus 7 Connex Engine cycles. Furthermore, no preprocessing of the data is required. The space requirement is $2N$, one copy of the string be compressed in RAM, and one copy in the connex engine.

1 Introduction

This paper presents a simple implementation of the Ziv-Lempel [9, 10] compression algorithm on the Connex Engine [6, 8], an in-memory device where memory cells can perform basic symbol comparisons and are connected together in a shift register fashion, offering a parallelism of computation of the lowest granularity.

The Connex Engine (CE) stores the string of symbols already processed by the algorithm and forms a natural dictionary, requiring only one cell per symbol of the original string, and where searching for prefixes is done in linear time complexity, one access to the CE per symbol searched. In this paper we assume that the size of the input alphabet is small enough that symbols can be stored in the individual cells of the CE.

The Connex Engine is a hierarchical in-memory device that permits fast string operations. It is typically attached to a processor as specialized memory circuit, or to a dedicated controller. It contains standard RAM circuit at the higher level of the hierarchy, and a specialized memory circuit at the lower level, the Connex Memory, that permits parallel search at the memory-cell level and shift operations. A controller oversees the
exchange of data between the two levels. Just as regular memory circuits, the operations supported by the Connex Engine can be performed in well-defined cycles whose duration is controlled by the current memory technology, which in today’s technology is in the 10 ns range. We refer to this cycle as connex cycle.

The Connex Engine can find all the locations of a given symbol in the Connex Memory in one connex cycle. Furthermore, it maintains all the symbols in a connected string fashion, such that any symbol can be inserted or deleted from any cell of the Connex Memory while maintaining the relative order of all the symbols, and this in one connex cycle only.

Our implementation of the Ziv-Lempel algorithm uses regular RAM to store the string to be compressed, and the CE to hold the uncompressed part of the string already scanned by the algorithm. Because string searches in the CE can be performed in one cycle per symbol searched, prefixes can be found extremely fast. Our implementation of the algorithm in C++ shows that compression can be performed at a rate of less than 50 processor cycles plus 7 connex engine access cycles per symbol. This translates in approximately 7 Msymbol/sec with today’s 1GHz processors. This speed of compression on a general-purpose string processing circuit compares favorably with other parallel VLSI solutions implementing Ziv-Lempel algorithm, such as the dedicated circuit of Jung and Burleson [5] compressing 12 M-symbols/sec with 1.2 μ CMOS technology.

In [6] we tested our design by implementing the Connex Engine circuit in a Xilinx FPGA, using a two-dimensional array to store the array of connex memory cells, and making our circuit ideally suited for VLSI implementation. This removes the limit that one-dimensional VLSI implementations previously carried [3].

The next section presents the general Zif-Lempel algorithm, and then its implementation on a computer equipped with a Connex Engine. We develop the complexity analysis of this parallel implementation in Section 5. Section 6 concludes this paper.

2 Basic Operations of the Connex Engine

The algorithm exploits the features of the Connex Engine (CE) as a hardware string-search engine that can perform fast string operations. To better understand the algorithm and its performance, we first present three of the commands supported by the Connex Engine, and which are heavily used by our algorithm. For simplicity, we assume here that the CE contains only connex memory and no standard RAM storage.

2.1 Basic search operations in the Connex Engine

The first command is find(x), where x is a symbol. Assume that the CE contains the symbols forming the string “cool cats can!” as shown in Figure 1.

Assume that the command find(c) is now issued to the CE. In our context, issuing a command means
that the symbol is fed to the device over a bus, along with the activation of hardware signals identifying this command, just as a symbol would be presented to a memory for a memory-write operation. In one clock cycle, the symbol \( c \) is broadcast to all the cells and their contents compared to it. When a match takes place, a bit called the marker is set in all the cells following the cell where the match occurred, as shown in Figure 2. We indicate that a cell has a marker set by putting a dot above the character it contains.

Figure 2: The CE after \texttt{find}(c).

Observe that \( o \) and the two \( a \)-symbols get their markers set because they all follow \( c \) in the string stored in the CE. The purpose of having a marker in each cell becomes evident when we introduce the next CE command: \texttt{conditional-find}(x), or \texttt{c-find}(x) for short, which also takes a symbol \( x \) as argument. During the execution of a \texttt{c-find} command, only the cells that have a marker already set perform a comparison. The others do not. Assuming that the contents of the CE is that shown in Figure 2, then applying \texttt{c-find}(a) to the CE changes its contents to that shown in Figure 3.

Figure 3: The CE after \texttt{c-find}(a).

Observe now that \( t \), and \( n \) are marked, because only two marked cells contained the symbol searched, \( a \), and, as a result, the two cells that follow these two \( a \)-cells get their markers set. The CE circuit is engineered such that a 1-bit signal is output to the controlling entity, such as a processor, to indicate if 1 or more markers were set by the last \texttt{find} or \texttt{c-find} operation. If we continue with \texttt{c-find}(t), then the CE gets set to the state illustrated in Figure 4.

Figure 4: The CE after \texttt{c-find}(t).

What was actually just performed is a search of the string “cat” in the CE: \texttt{find}(c), \texttt{c-find}(a), and \texttt{c-find}(t). Note that this search required three commands, and thus takes only 3 memory cycles, one cycle for each symbols in the query string, independently of the length of the string stored in the CE. If the CE had contained several instances of “cat”, then all would have been found and marked. Note furthermore
that no pre-processing of the data is required, only initially storing the string in the CE, done by shifting the string into the CE, one symbol at a time, and requiring 1 memory cycle per symbol. As a result, searching for a substring of $N$ symbols in a string of $M$ symbols stored in the CE requires exactly $N$ cycles. The CE supports forward string searches, as presented above, as well as backward, or reverse string searches. In this case, looking for the substring “cat” would require the execution of \texttt{reverse-find(t)}, \texttt{reverse-cfind(a)}, and \texttt{reverse-cfind(c)}.

### 2.2 One-cycle Insertions in the Connex Engine

\texttt{Insert(c)} is another CE command heavily used by the algorithm, and it operates on the left-most marked symbol stored in the CE. Assuming that the contents of the CE are those shown in Figure 2, that is all the symbols following $c$ are marked, then the result of issuing the command \texttt{insert(z)} is illustrated in Figure 5.

![Figure 5: The CE after insert(z).](image)

In one cycle, the contents and markers of all the cells to the right of left-most marked cell, and including it, are shifted right, and the new character is inserted in the cell originally containing the left-most marker.

These three CE commands become pillar operations in our implementation of the Ziv-Lempel algorithm, which we present in its general form next, and then develop in the context of the Connex Engine circuit.

### 3 The general Ziv-Lempel algorithm

The Ziv-Lempel algorithm [9, 10] is the basis for many compression software, including the Unix compress utility. In this paper we use the original definition of the algorithm. While modifications and improvement of the algorithm have been presented and implemented before, such as LZSS in [7], or LZB in [1], for example, our aim here is to evaluate the design and performance of the original compression algorithm with our new device, not to achieve the best performance.

Two definitions are necessary before developing the algorithm [4]:

**Definition** For any position $i$ in a string $V$ of length $N$, define the substring $\text{Prior}_i$ to be the longest prefix of $V[i..N]$ that also occurs as a substring of $V[1..i-1]$.

**Definition** For any position $i$ in $V$, define $L_i$ as the length of $\text{Prior}_i$. For $L_i > 0$, define $S_i$ as the starting position of the left-most copy of $\text{Prior}_i$. 
Assuming that the Ziv-Lempel algorithm has just output the compressed version of \( V[1..i-1] \), and is looking at the new symbol in \( V[i] \), it starts by finding \( \text{Prior}_i \), that is the longest prefix of \( V[i..N] \) that occurs as a substring in \( V[1..i-1] \), and records its starting point \( S_i \) in \( V[1..i-1] \), along with its length \( L_i \). It then outputs \( (S_i, L_i) \) as the compressed version of \( \text{Prior}_i \), followed by \( x \), the next character following \( \text{Prior}_i \) in \( V[i..N] \). Before iterating again, the algorithm updates \( i \) by adding to it \( L_i \), so that it starts its loop again at \( V[i] \), the symbol following \( x \) in \( V \).

4 Ziv-Lempel on the Connex Engine

Porting the Ziv-Lempel compression algorithm to the Connex Engine is straightforward. Figure 6, 7, and 8 illustrate the concept. The string \( V \) of \( N \) symbols is stored in conventional RAM shown on the right hand-side of Figure 6. The algorithm has finished compressing the first \( i-1 \) symbols (shaded part of the RAM) and is about to start on the \( i^{th} \) symbol. The \( i-1 \) symbols are replicated in the CE shown on the left-hand side of the figure. A special symbol \# is used as a sentinel in the CE to mark the end of the prefix.

![Figure 6: The contents of the connex memory and of the Ram when the algorithm is ready to start processing character V[i].](image)

The algorithm uses the \texttt{find} and \texttt{c-find} commands to locates the longest prefix \( \{V[i], V[i+1], V[i+2]...\} \) present in the CE, and represented in Figure 7 as the blue block starting at CE Index \( S \), and containing \( L \) symbols.

The algorithm then outputs \( (S, L) \) followed by the symbol \( x \) which follows the prefix just found in \( V \). The collection of \( x \)-symbols and \( (S, L) \) pairs output this way represents the compressed version of \( V \).

Using the CE command \texttt{find(#)}, the algorithm marks the end of the part of \( V \) currently in the CE, and, using the \texttt{insert} command, appends to it the prefix just found, followed by \( x \).

The algorithm updates its running index \( i \) to point to the symbol following \( x \) in \( V \), and we are in the situation illustrated in Figure 8, that is the same computational state we were in at the beginning of the loop, and illustrated in Figure 6.

The algorithm continues until it outputs a final \( (S, L) \) pair or a final symbol \( x \).
Figure 7: The contents of the connex memory and the Ram when the longest prefix (in blue) starting with $V[i]$ has been located in the CE.

Figure 8: The contents of the connex memory and the Ram when the prefix found has been inserted in the CE, and the index $i$ incremented.
5 Complexity Analysis

A description of the CE-based algorithm compressing the string $V$ of $N$ symbols follows:

Step 1: $i=0$, CE="#"

Step 2: find longest prefix in CE matching $V[i..i+L-1]$

Step 3: compute $S$ as the starting location of this prefix in the Connex Engine CE.

Step 4:

if ($L==0$) then
  output($V[i]$)
else
  output(($S,L)V[i+L]$)
  $i = i + L + 1$

Step 5: $i = i + 1$

Step 6: mark symbol # in CE, and insert $V[i..i+L]$ in CE

Step 7: if ($i < N$) go back to Step 2 else stop.

The complexity of Step 2 is $O(L)$, since it requires $L + 1$ connex cycles to find the longest substring of $L$ symbols in the Connex Engine: 1 cycle for the first find, $L - 1$ successful c-find commands, followed by one unsuccessful c-find.

The complexity of Step 3 is $O(1)$. The Connex Engine outputs the linear address of the left-most marked symbol in permanence. To find $S$ requires reading this status information and subtracting $L$ from it.

Step 4 has complexity $O(L)$ because it outputs $L + 1$ symbols.

Finally, Step 6 requires two connex cycles to find the #-symbol and move the marker one position to its left, and $L$ connex cycles to append the string just found to the contents of the CE with the insert command, for an overall $O(L)$ complexity.

The global complexity of the seven steps is clearly $O(N)$, where $N$ is the number of symbols in the string to be compressed.

To compute a precise estimate of the multiplicative constant hidden in the big-Oh notation, and to validate the algorithm we implemented the algorithm in C++, and its code is given in the appendix. By counting the number of assignments, arithmetic operations, jumps, tests, and output operations, as well as connex-engine operations, we get a precise estimate of the linear equation controlling the execution time of the algorithm. Table 1 presents the notation we use to represent these different operations. With the
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{CE}$</td>
<td>connex cycle</td>
</tr>
<tr>
<td>$T_a$</td>
<td>integer addition</td>
</tr>
<tr>
<td>$T_c$</td>
<td>integer comparison</td>
</tr>
<tr>
<td>$T_i$</td>
<td>assignment</td>
</tr>
<tr>
<td>$T_j$</td>
<td>jump</td>
</tr>
<tr>
<td>$T_{out}$</td>
<td>output one symbol</td>
</tr>
<tr>
<td>$T_r$</td>
<td>indexing in 1 dim array</td>
</tr>
</tbody>
</table>

Table 1: description of various elementary cycles

The exception of $T_{out}$ and $T_{CE}$ which represent the time required to output a symbol to a file, and that to execute a CE operation, all the times used correspond typically to processor instructions, which we assume all execute in constant time.

Table 2 present an account of the total number of basic operations executed in each phase of the C++ algorithm. $L$ represents the size of the prefix found.

<table>
<thead>
<tr>
<th>Phases</th>
<th>Complexity</th>
<th>Basic Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Initialization</td>
<td>$O(1)$</td>
<td>$3T_i + 2T_{CE}$</td>
</tr>
<tr>
<td>2 Find longest prefix</td>
<td>$O(L)$</td>
<td>$5T_i + T_a + L(4T_i + 2T_a + T_r + 3T_c + 4T_{CE} + 2T_j)$</td>
</tr>
<tr>
<td>3 Output</td>
<td>$O(1)$</td>
<td>$T_a + 5T_{out}$</td>
</tr>
<tr>
<td>4 Concatenate prefix in CE</td>
<td>$O(L)$</td>
<td>$2T_{CE} + T_i + L(T_i + 2T_a + T_c + T_r + T_{CE})$</td>
</tr>
<tr>
<td>5 Advance loop</td>
<td>$O(1)$</td>
<td>$T_i + T_a + T_j$</td>
</tr>
</tbody>
</table>

Table 2: Number of cycles taken by the C++ LZ-CE algorithm

For simplicity, and while maintaining good accuracy, we can assume that $T_a = T_c = T_i = T_j = T_r = T$, the equivalent of one CPU cycle. Then the overall equation for the number of cycles required to compress a string of $N$ symbols on the Connex Engine becomes:

$$\text{Number of cycles} = 5T + P(11T + 5T_{out} + 2T_{CE}) + N(17T + 5T_{CE})$$

where $P$ is the number of $x$ or $(S, L)x$ terms output by the algorithm, which is also the number of prefixes found, including 0-length prefixes. Since clearly $P \leq N$, we can bound the number of cycles taken by the algorithm by $5T + N(28T + 5T_{out} + 7T_{CE})$.

In summary, the overhead per symbol compressed is very small, and equal to 28 CPU cycles, plus 7 Connex Engine cycles, plus the time required to output 5 symbols and/or numbers. If the symbols are stored in an array instead of output to a file, the overhead is less than 50 CPU cycles plus 7 Connex Engine
cycles per symbol compressed. Since the connex engine is implemented using standard memory technology, one can expect a ratio of CPU cycle to Connex Engine cycle in the range of 1:10 to 1:50. Compression rates of up to 7 MB/sec on a serial 1 GHz processor with an attached Connex Engine circuit are thus possible.

The overall space requirement for the execution of the algorithm is one standard RAM cell plus one Connex Engine cell per symbol.

6 Conclusion

The connex engine is a general purpose string processing device, and it is well suited for compression applications such as the Ziv-Lempel algorithm or its derivatives, where it competes well with dedicated VLSI implementations.

Its ability to perform symbol searches and symbol-insert operations in one cycle has several advantages for the Ziv-Lempel algorithm:

- strings can be searched in time proportional to the size of the string, one CE cycle plus a few CPU cycles per symbol.
- the dictionary of prefixes is trivially implemented in the CE as a list of symbols already scanned, requiring no special data structures and no preprocessing.
- the lack of dedicated data structure keeps the memory utilization optimum.

The problem of limited space which affects the management of dictionaries [2] used in the serial implementation of the Ziv-Lempel algorithm has a simple and natural solution in our case. When the size of the string already processed becomes larger than the size of the CE, the insertion of the newly found prefixes in the CE automatically shifts out the oldest seen symbols currently in the CE. This way a CE of size $M$ automatically maintains the last $M - 1$ symbols (remember that we use a sentinel #) scanned by the algorithm, and prefixes are thus searched among the last $M - 1$ symbols seen by the algorithm. No overhead is incurred by this size limitation.

The domain of application for the Connex Engine is rich, with applications in databases, bioinformatics [8], and where string processing is paramount. The hierarchical design of the circuit balances its performance and cost, and its two-dimensional VLSI implementation allows a high level of integration.

References


**Appendix**

This appendix contains the C++ code used to test the algorithm. The CE operations are implemented by in-line functions adorning a "CE_" prefix.

```cpp
int i = 0, j, S, lastS, L;
char x;

CE_init("#");
do {
    // find longest prefix in RAM that matches in CE
```
L=0; j=i; lastS=0;
do {
    i=j+ L++;
    x=V[i];
    if (L==1)
        CE_find(c);
    else
        CE_cFind(c);
    lastS=S;
    S=CE_index()-L;
} while (CE_markers() && (i<N-1));
L--;
S=lastS;

// prefix of length more than 1?
if (L!=0)
    cout << "(" << S+1 << "," << L << ")";
    cout << x;

// find end of string in connex to add suffix
// just found
CE_Find('#');
CE_moveMarker(); // shift the marker left

for (int k=S; k<S+L; k++)
    CE_insert(V[k]);
    CE_insert(c);

// move on to next char
i++;
} while (i<N);