Accelerating the CKY Parsing Using FPGAs ⋆

Jacir L. Bordim, Yasuaki Ito, and Koji Nakano

School of Information Science
Japan Advanced Institute of Science and Technology
1-1, Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan
(bordim,yasuaki,knakano)@jaist.ac.jp

Abstract. The main contribution of this paper is to present an FPGA-based implementation of an instance-specific hardware which accelerates the CKY (*Cook-Kasami-Younger*) parsing for context-free grammars. Given a context-free grammar G and a string x, the CKY parsing determines if G derives x. We have developed a hardware generator that creates a Verilog HDL source to perform the CKY parsing for any given context-free grammar G. The created source is embedded in an FPGA using the design software provided by the FPGA vendor. We evaluated the instance-specific hardware, generated by our hardware generator, using a timing analyzer and tested it using the Altera FPGAs. The generated hardware attains a speed-up factor of approximately 750 over the software CKY parsing algorithm. Hence, we believe that our approach is a promising solution for the CKY parsing.

1 Introduction

An FPGA (Field Programmable Gate Array) is a programmable VLSI in which a hardware designed by users can be embedded instantly. Typical FPGAs consist of an array of programmable logic elements, distributed memory blocks, and programmable interconnections between them. Our goal is to use the FPGAs to accelerate useful computations. In particular, the challenge is to develop FPGA-based solutions which are faster and more efficient than traditional software approaches.

Our basic idea for accelerating computations using the FPGAs is inspired by the notion of *partial computation* [10]. Let $f(x, y)$ be a function to be evaluated in order to solve a given problem. Note that such a function might be repeatedly evaluated only for a fixed x. When this is the case, the computation of $f(x, y)$ can be simplified by evaluating an instance-specific function f_x such that $f_x(y) = f(x, y)$. For example, imagine a problem such that an algorithm to solve it evaluates $f(x, y) = x^3 + x^2y + y$ repeatedly. If $f(x, y)$ is evaluated only for $x = 2$, then the formula can be simplified such that $f_2(y) = 8 + 5y$. The optimization of function f_x for a particular x is called a *partial computation*. Usually, a partial computation has been used for optimizing a function f_x in the context of software, *i.e.*, sequential programs [8]. Our novel idea is to built a

⋆ Work supported in part by the Ministry of Education, Science, Sports, and Culture, Government of Japan, Grant-in-Aid for Exploratory Research (90113133).

hardware that is optimized to compute $f_x(y)$ for a fixed x and various y. More specifically, our goal is to present an FPGA-based instance-specific solution for problems that involves a function evaluation for $f(x,y)$ satisfying the following properties: (i) the value of a fixed instance x depends on the instance of the problem, and (ii) the value of $f(x,y)$ is repeatedly evaluated for various y to solve the problem. The FPGA-based instance-specific solution that we propose evaluates $f_x(y)$ $(= f(x,y))$ using a hardware for function f_x. If the problem we need to solve satisfies these properties, it is worth attempting the instance-specific solution.

The main contribution of this paper is to present an instance-specific hardware which accelerates the parsing for context-free grammars [12] using the FPGA-based approach described above. Let $f(G,x)$ be a function such that G is a context grammar, x is a string, and $f(G,x)$ returns a Boolean value such that $f(G,x)$ returns TRUE iff G derives x. It is well-known that the *CKY(Cook-Kasami-Younger) parsing* [1] computes $f(G,x)$ in $O(n^3)$ time, where n is the length of x [1]. The parsing of context-free languages has many application in various areas including natural language processing [5, 14], compiler construction [1], informatics [13], among others.

Several studies have been devoted to accelerate the parsing of context-free languages [4, 9, 11, 14]. It has been shown that the parsing for a string of length n can be done in $O((\log n)^2)$ time using n^6 processors on the PRAM [9]. Also, using the mesh-connected processor arrays, the parsing can be done in $O(n^2)$ time using n processors as well as in $O(n)$ time using n^2 processors [11]. Since these parallel algorithms need at least n processors, they are unrealistic for large n. Ciressan *et al.* [6, 7] have presented a hardware for the CKY parsing for a restricted class of context-free grammar and have tested it using FPGA. However, the hardware design and the control algorithm are essentially the same as those on the mesh-connected processors [11], and they are not instance-specific.

For the purpose of instance-specific solution for parsing context-free languages, we present a hardware generator that produces a Verilog HDL source that performs the CKY parsing for any given context-free grammar G. The key ingredient of the produced design is a hardware component to compute a binary operator $\otimes_G$ such that $2^N \times 2^N \to 2^N$, where N is the set of non-terminal symbols in G. More specifically, let U and V be a set of non-terminals in G that derive strings α and β, respectively. The operator $U \otimes_G V$ returns the set of non-terminals that derive $\alpha\beta$ (i.e. the concatenation of α and β). The CKY parsing algorithm repeats the evaluation of $\otimes_G$ for $O(n^3)$ times. The details of $\otimes_G$ will be explained in Section 2. Our hardware generator provides two types of hardware. The first hardware has one component for computing $\otimes_G$. The second one has two or more components to further accelerate the table algorithm for the CKY parsing.

The generated Verilog HDL source is compiled using the Altera Quartus II design tool [3], and the object file obtained is downloaded into the Altera APEX20K series FPGAs [2]. The programmed FPGA compute $f_G(x)$, i.e. determines if G derives x for a given string x. Figure 1 illustrates our hardware CKY parsing system. Given strings $x_1, x_2, x_3, \ldots$ by the host PC, the FPGA computes and returns $f_G(x_1), f_G(x_2), f_G(x_3), \ldots$ to the host.

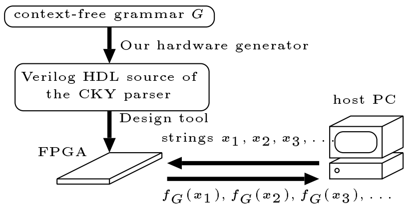


Fig. 1. Our hardware parsing system

From the theoretical point of view, our instance-specific solution is much faster than the software solutions. To clarify how our solution accelerates the CKY parsing, we provide the following two software approaches as counterparts:

Naive algorithm: This algorithm computes $\otimes_G$ by checking all p production rules in $O(p)$ time. The CKY parsing using the naive algorithm runs in $O(n^3 p)$ time.

Table algorithm: This algorithm computes $\otimes_G$ by looking up $(\frac{b}{c})^2$ tables of 2^{2c} words with b bits in $O((\frac{b}{c})^2)$ time, where b is the number of non-terminal symbols in G. Although c can take any integer, in practice, c does not exceed 16, possibly $c \leq 8$. The CKY parsing using the table algorithm runs in $O(n^3(\frac{b}{c})^2)$ time.

Our instance-specific solution evaluates $\otimes_G$ in $O(\log b)$ time and the CKY parsing using this approach runs in $O(n^3 \log b)$ time. Since $b \leq p$ always hold, our solution is faster than these software approaches from theoretical point of view.

We have evaluated the performance of our instance-specific solution using the timing analyzer of Quartus II and test it using an APEX20K series FPGA. In order to evaluate the performance of our instance-specific solution, we also have implemented the above software solutions and measured the performance using a Pentium4-based PC.

The timing analysis results show that our instance-specific hardware attains up to 750 speed-up factor over the software solutions. Thus, we strongly believe that our approach for parsing context-free languages is a promising solution.

2 The CKY parsing and software solutions

The main purpose of this section is to briefly describe the CKY parsing and show two software solutions. Let $G = (N, \Sigma, P, S)$ denote a *context-free grammar* such that N is a set of non-terminal symbols, Σ is a set of terminal symbols, P is a set of production rules, and S ($\in N$) is the start symbol. A context-free grammar is said to be a *Chomsky Normal Form* (CNF), if every production rule in P is in either form $A \to BC$ or $A \to a$, where A, B, and C are non-terminal symbols and a is a terminal symbol.

We are interested in the parsing problem for a context-free grammar with CNF. More specifically, for a given CNF context-free grammar G and a string x over Σ, the parsing problem asks to determine if the start symbol S derives x. For example, let $G_{\text{example}} = (N, \Sigma, P, S)$ be a grammar such that $N = \{S, A, B\}$,

$\Sigma = \{a, b\}$, and $P = \{S \to AB, S \to BA, S \to SS, A \to AB, B \to BA, A \to a, B \to b\}$. Context grammar G derives $abaab$, because S derives it as follows:

$$S \Rightarrow AB \Rightarrow ABA \Rightarrow ABAA \Rightarrow ABAAB \Rightarrow \cdots \Rightarrow abaab.$$

We are going to explain the *CKY parsing scheme* that determines whether G derives x for a CNF context-free grammar G and a string x. Let $x = x_1 x_2 \cdots x_n$ be a string of length n, where each x_i $(1 \leq i \leq n)$ is in Σ. Let $N[i, j]$ $(1 \leq i \leq j \leq n)$ denote a subset of N such that every A in $N[i, j]$ derives a substring $x_i x_{i+1} \cdots x_j$. The idea of the CKY parsing is to compute every $N[i, j]$ using the following relations:

$$N[i,i] = \{A \mid (A \to x_i) \in P\}$$

$$N[i,j] = \bigcup_{k=i}^{j-1} \{A \mid (A \to BC) \in P, B \in N[i,k], \text{ and } C \in N[k+1,j]\}$$

A two-dimensional array N is called the *CKY table*. A grammar G generates a string x iff S is in $N[1, n]$. Let $\otimes_G$ denote a binary operator $2^N \times 2^N \to 2^N$ such that $U \otimes_G V = \{A \mid (A \to BC) \in P, B \in U, \text{ and } C \in V\}$. The details of the CKY parsing are spelled out as follows:

CKY parsing

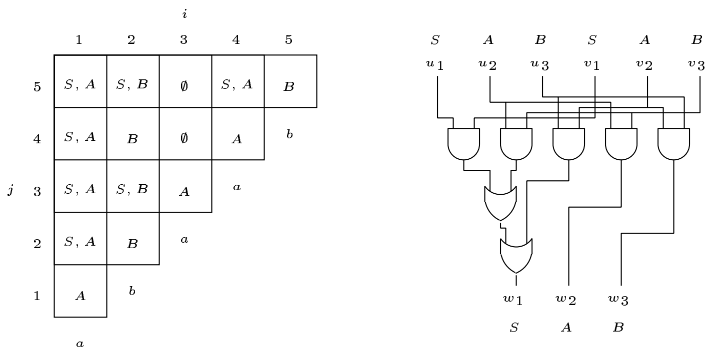
```
1. N[i,i] ← {A | (A → x_i) ∈ P} for every i (1 ≤ i ≤ n)
2. N[i,j] ← ∅ for every i and j (1 ≤ i < j ≤ n)
3. for j ← 2 to n do
4.     for i ← j − 1 downto 1 do
5.         for k ← i to j − 1 do
6.             N[i,j] ← N[i,j] ⋃(N[i,k] ⊗_G N[k+1,j])
```

The first two lines initialize the CKY table, and the next four lines compute the CKY table. Figure 2(a) illustrates the CKY table for G_{example} and the string $abaab$. Since $S \in N[1, 5]$, one can see that G_{example} derives $abaab$. Clearly, the last four lines are dominant in the CKY parsing. Let T be the computing time necessary to perform an iteration of the line 6. Then, line 6 is executed for

$$\sum_{j=2}^{n-1} \sum_{i=1}^{j-1} \sum_{k=i}^{j-1} T = T \sum_{j=2}^{n-1} \sum_{i=1}^{j-1} (j - i) = \frac{1}{6} T(n^3 - 3n^2 + 2n)$$

times. Let us evaluate the computing time T necessary to perform line 6, i.e., necessary to evaluate a binary operator $\otimes_G$. We will present two approaches that compute $U \otimes_G V$ by sequential (software) algorithms for any given U and V.

In the first approach, named *naive algorithm*, it is checked whether $B \in U$ and $C \in V$ for every production rule $A \to BC$ in P. Clearly, using a reasonable data structure, this can be done in $O(1)$ time. Hence, $U \otimes_G V$ can be evaluated in $O(p)$ time, where p is the number of production rules in P that has the form $A \to BC$. Thus, the first approach enables us to perform the CKY parsing in $O(n^3 p)$ time.



(a) (b)

Fig. 2. (a) The CKY table for G_{example} and *abaab*. (b) The circuit for computing $\otimes_{G_{\text{example}}}$.

Suppose that N has b non-terminal symbols, and let $N = \{N_1, N_2, \ldots, N_b\}$. The second approach that we call *table algorithm* uses a huge look-up table that stores the values of $U \otimes_G V$ for every pair U and V. For a given U $(\in 2^N)$, let $u_1 u_2 \cdots u_b$ be the b-bit vector such that $u_i = 1$ iff $N_i \in U$ for every i $(1 \le i \le b)$. Similarly, let $v_1 v_2 \cdots v_b$ be the b-bit vector for V $(\in 2^N)$. For the purpose of computing $U \otimes_G V$, we use a look-up table of $2^{2b} \times b$ in memory (i.e., the address and the data are $2b$ bits and b bits, respectively). The $u_1 u_2 \cdots u_b v_1 v_2 \cdots v_b$-th entry of the table stores $w_1 w_2 \cdots w_b$, where $w_1 w_2 \cdots w_b$ is the b-bit vector representation of $W = U \otimes_G V$. Clearly, if such table is available, $U \otimes_G V$ can be computed in $O(1)$ time. However, the table can be too large even if b is not large. If P has $b = 64$ non-terminal symbols, then the table must have $2^{2 \cdot 64} \times 64 = 2^{134} \approx 10^{40}$ bits, which is too large for practical purposes.

We will modify the table algorithm to reduce the table size. Let us partition N into equal-sized subsets such that $N^i = \{N^{c(i-1)+1}, N^{c(i-1)+2}, \ldots, N^{ci}\}$ $(1 \le i \le \frac{b}{c})$. We use $(\frac{b}{c})^2$ binary operators $\otimes_G^{i,j}$ $(1 \le i, j \le \frac{b}{c})$ such that

- $\otimes_G^{i,j}$ is $2^{N^i} \times 2^{N^j} \to 2^N$, and
- $(U \cap N^i) \otimes_G^{i,j} (V \cap N^j) = \{A \mid (A \to BC) \in P, B \in U \cap N^i, \text{ and } C \in V \cap N^j\}$.

It is easy to see that,

$$U \otimes_G V = \bigcup_{1 \le i,j \le (\frac{b}{c})^2} (U \cap N^i) \otimes_G^{i,j} (V \cap N^j).$$

Thus, by evaluating $\otimes_G^{i,j}$ for every pair i and j, we can compute $\otimes_G$. As before, $\otimes_G^{i,j}$ can be computed by looking up a table of size $2^{2c} \times b$. Hence, $\otimes_G$ can be computed in $O((\frac{b}{c})^2)$ time by looking up $(\frac{b}{c})^2$ tables. The total size of the tables is $\frac{b^3}{c^2} 2^{2c}$ bits. If $b = 64$ and $c = 8$, then the tables should have $2^{28} = 256M$ bits, which is feasible. However, we need to look up the table for $(\frac{b}{c})^2 = 64$ times. Note that the size of the tables and the number of times needed to be looked up are independent of the number p of production rules. Thus, the second approach is more efficient for large p.

3 Our instance-specific hardware for the CKY parsing

This section is devoted to show our instance-specific hardware for the CKY parsing. We first accelerate the evaluation of $\otimes_G$ by building a circuit for computing $\otimes_G$ in an FPGA. We then go on to show the hardware details to build this circuit.

Recall that each U and V ($\in 2^N$) are represented by b-bit binary vectors $u_1u_2\cdots u_b$ and $v_1v_2\cdots v_b$, respectively. Our goal is to compute the vector $w_1w_2\cdots w_b$, which represents $W = U \otimes_G V$. For a particular w_k, we are going to show how w_k is computed. Let $N_k \rightarrow N_{i_1}N_{j_1}$, $N_k \rightarrow N_{i_2}N_{j_2}$, $\ldots$, and, $N_k \rightarrow N_{i_s}N_{j_s}$ be the production rules in P whose non-terminal in the left-hand side is N_k. Then, w_k is computed by the following formula:

$$w_k = (v_{i_1} \wedge u_{j_1}) \vee (v_{i_2} \wedge u_{j_2}) \vee \cdots \vee (v_{i_s} \wedge u_{j_s}).$$

Thus, w_k can be computed by a combinatorial circuit using s AND-gates and $s-1$ OR-gates with fan-in 2. Furthermore, the depth of the circuit (or the maximum number of gates over all paths in the circuit) is $\lceil \log(s-1) \rceil + 1$. Since we have p production rules of the type $A \rightarrow BC$ in P, then $w_1w_2\cdots w_b$ can be computed by a circuit with p AND-gates and $p-b$ OR-gates. Because $s \leq b^2$ always hold, the depth of the circuit is no more than $\lceil \log(b^2-1) \rceil + 1 \leq 2\log b + 1$. Thus, the CKY parsing can be done in $O(n^3 \log b)$ time using this circuit. Figure 2(b) illustrates a circuit for $\otimes_{G_{\mathrm{example}}}$. Since G_{example} has 5 production rules and 3 non-terminal symbols, the circuit has 5 AND gates and $5-3=2$ OR gates.

The sequential algorithms we have shown in Section 2 take $O(p)$ time or $O((\frac{b}{c})^2)$ time to evaluate $\otimes_G$. On the other hand, our circuit for $\otimes_G$ has the delay time proportional to $O(\log b)$. Since $b \leq p \leq b^3$ always holds, the circuit for $\otimes_G$ is faster than the sequential algorithms from the theoretical point of view. In what follows, we are going to show the implementation details of our instance-specific hardware. Our first hardware implementation of the CKY parsing uses the following basic components: (*i*) a b-bit n^2-word (dual-port) memory; (*ii*) a b-bit n-word (dual-port) memory; (*iii*) a CKY circuit for $\otimes_G$; (*iv*) an array of b OR gates; and (*v*) a b-bit register.

Figure 3(a) illustrates our first implementation for the CKY parsing. The b-bit n^2-word memory stores the CKY table. The input, $N[1,1], N[2,2], \ldots, N[n,n]$ is supplied to the b-bit n^2-word memory. The b-bit n-word memory stores a row of the CKY table that is being processed. In other words, it stores the j-th row $N[1,j], N[2,j], \ldots$ of the CKY table, where j is the variable appearing in line 3 of the CKY parsing. The b-bit register stores the current value of $N[i,j]$, which is computed in line 6 of the CKY parsing. The array of b OR gates is used to compute "$\bigcup$" in line 6. The b-bit n^2-word memory supplies the b-bit vector representing $N[i,k]$ to the CKY circuit. Similarly, the b-bit n-word memory outputs the b-bit vector for $N[k+1,j]$. The CKY circuit receives them and computes the b-bit vector for $N[i,k] \otimes_G N[k+1,j]$. Using this hardware implementation,

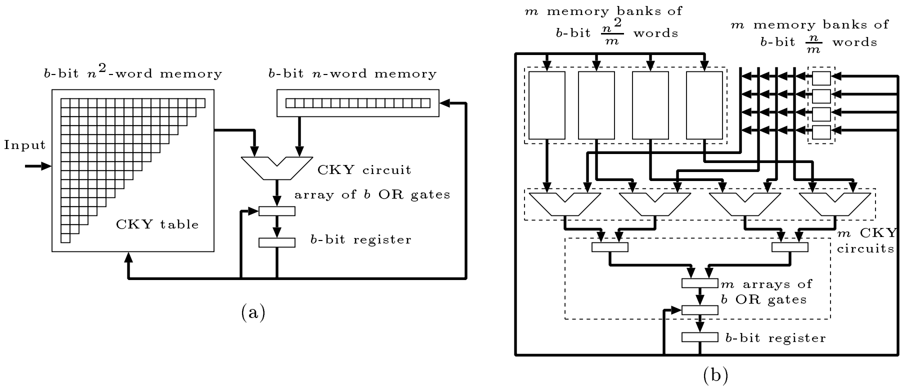


Fig. 3. (a) A hardware implementation for the CKY parsing. (b) Our parallel implementation of the CKY parsing for $m = 4$.

line 6 of the CKY parsing is computed in a clock cycle. Thus, the CKY parsing can be done in n^3 clock cycles. Furthermore, in a real implementation, a clock cycle is proportional to $O(\log b)$. Thus, the computing time is $O(n^3 \log b)$.

We are going to parallelize the CKY parsing using two or more CKY circuits. For this purpose, we partition the CKY table into m subtables $S(0)$, $S(1)$, $\ldots, S(m-1)$ such that $S(l)$ is storing $N[i,j]$ satisfying $j - i \bmod m = l$. Clearly, for any m consecutive elements $N[i,k], N[i,k+1], \ldots, N[i,k+m-1]$ in a column of the CKY table, these elements are stored in distinct subtables. Thus, the consecutive m elements can be accessed in the same time if each subtable is stored in a memory bank. This fact allows us to parallelize the CKY parsing using m CKY circuits. In order to evaluate the performance of the above approaches, we have implemented the instance-specific hardware CKY parser using a single CKY circuit (*single-circuit*), two CKY circuits (*double-circuit*), and four CKY circuits (*quad-circuit*).

Our parallel implementation of the CKY parsing uses the following basic components: (*i*) m (dual-port) memory banks of b-bit $\frac{n^2}{m}$ words; (*ii*) m (dual-port) memory banks of b-bit $\frac{n}{m}$ words; (*iii*) m CKY circuits for $\otimes_G$; (*iv*) m arrays of b OR gates; and (*v*) a b-bit register. Figure 3(b) illustrates our parallel implementation for the CKY parsing. The m memory banks of b-bit $\frac{n^2}{m}$ words are used to store m subtables, one bank for each subtable. Also, the m memory banks of b-bit $\frac{n}{m}$ words store a row of the CKY table that is currently being processed. When $N[i,j]$ is computed, these m memory banks are storing the j-th row $N[1,j], N[2,j], \ldots, N[j,j]$ of the CKY table. More precisely, $N[l+1,j], N[l+m+1,j], N[l+2m+1,j], \ldots$ are stored in the l-th bank ($0 \leq l \leq m$). Thus, m evaluations of $\otimes_G$, say, $N[1,1] \otimes_G N[2,j], N[1,2] \otimes_G N[3,j], \ldots, N[1,m] \otimes_G N[m+1,j]$, can be evaluated in a clock cycle because $N[1,1], N[2,j], N[1,2], N[3,j], \ldots, N[1,m], N[m+1,j]$ are stored in distinct memory banks. This allows us to accelerate the CKY parsing by a factor of m. Thus, the computing time for the CKY parsing is $O(\frac{n^3 \log b}{m})$ for $m \leq n$.

4 Performance evaluation

We have evaluated the performance our instance-specific solution using the timing analyzer of Quartus II and tested it using an APEX20K series FPGA(EP20K-400EBC652-1X, typical 400K gates with 200Kbits embedded memory and 16K logic elements). In order to evaluate the performance of our instance-specific solution, we have implemented two software solutions and measured the performance on a 1.7GHz Pentium4-PC using Linux OS (Kernel 2.4.9). More specifically, we first evaluate the performance of both software and hardware solutions to compute the function $\otimes_G$. Next, we show the performance evaluation for the CKY parsing algorithm.

Figure 4(a) shows the running time of our hardware and software implementations to compute the function $\otimes_G$. Note that a word of data on a Pentium-based PC is 32-bit. Thus, we have implemented the 32-bit vector using a single word and the 64-bit vector using two words. As a consequence, the two word implementation for the 64-bit vector adds an overhead which makes it slower than the 32-bit vector solution. Recall that the naive algorithm checks whether or not $B \in U$ and $C \in V$ for every production rule $A \rightarrow BC$ in P. Hence, the computing time of the naive algorithm is proportional to the number of production rules.

As for the table algorithm, the computing time obeys a more regular pattern since the running time does not depend on the number of rules but rather it depends on the number of times it has to access the table. Recall that the table algorithm has to perform $(\frac{b}{c})^2$ table look-ups for b non-terminal symbols to compute $\otimes_G$. Thus, by increasing the value of b, the running time of the table algorithm also increases. As expected, for small values of p, the running time of the naive algorithm beats the table algorithm. However, as the number of p increases, the table algorithm is much faster than the naive algorithm.

In computing the function $\otimes_G$, our hardware implementation attained a speed-up of nearly 1000 over the table algorithm, using 64-bit vector approach. A speed-up of nearly 100 is observed using 32-bit vector approach. Comparing the results with the naive algorithm, the gain is even more apparent: for $p = 16384$, our hardware implementation attained a speed-up of nearly $22,000$, over the naive algorithm using 64-bit vector approach, and a speed-up of nearly $7,300$ using 32-bit vector approach. Since the running time or our hardware implementation is independent of the number of encoding bits, the 32-bit vector and the 64-bit vector approaches have nearly the same running time.

Figure 4(b) shows the computing time of the CKY algorithm for $b = 32$ and $l = 32$ (where l represents the length of the input string). The figure shows the computing time for the sequential algorithms as well as for the hardware implementation. We have implemented the CKY algorithm in hardware using single-, double-, and quad-circuit and plotted the running times of both hardware and software approaches. The software solutions have followed the same pattern observed in Figure 4(a). This is due to the fact that a good portion of the computation time is spent evaluating $\otimes_G$. Our hardware implementation for single-circuit also follows from the previous figure. One can see that the double- and quad-circuit approaches indeed accelerate the CKY parsing as we have predicted in Section 3.

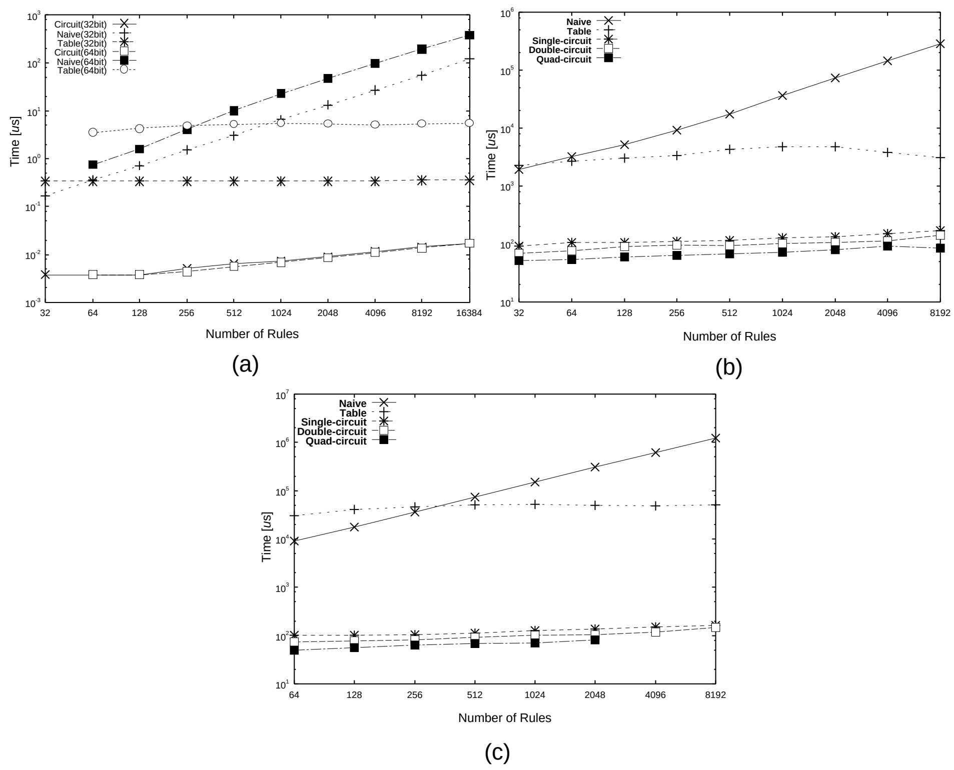


Fig. 4. (a) Computing time to evaluate $\otimes_G$. (b) Computing time of the CKY algorithm with $b = 32$ and $l = 32$. (c) Computing time of the CKY algorithm with $b = 64$ and $l = 32$.

Figure 4(c) shows the computing time of the CKY algorithm for $b = 64$ and $l = 32$. We observed that the running time follows the same pattern of the CKY algorithm for $b = 32$ and $l = 32$. As mentioned before, the 64-bit vector approach adds an extra overhead to the software solutions which does not occur on the hardware implementations. As a result, we observe a degradation on the running time of the software solutions. The number of logic elements necessary to compute $p = 2048$, using a quad-circuit, is nearly $9,600$. For $p = 4096$, the number of logic elements necessary to build the quad-circuit surpasses the overall number of logic elements provided by our FPGA. Because of this, we have been able to implement the quad-circuit for p up to 2048.

Table 1 shows the speed-up of the CKY algorithm over the table algorithm (software approach). The computing time of each algorithm can be seen in the previous figures. For $b = 32$ and $l = 32$, our hardware approach achieved speed-up of nearly: 40 using a single-circuit; 50 using a double-circuit; and 70 using a quad-circuit. Our results are even more appealing for $b = 64$ and $l = 32$. In this case, our hardware approach achieved speed-up of nearly: 460 using a single-circuit; 580 using a double-circuit; and 750 using a quad-circuit. Thus, from the above results, we argue that our hardware approach is indeed a promising solution to solve the CKY parsing.

	$b = 32, l = 32$			$b = 64, l = 32$		
p	Single	Double	Quad	Single	Double	Quad
32	25	33	44	–	–	–
64	25	35	50	304	419	611
128	29	34	51	395	519	731
256	30	35	53	441	577	730
512	37	46	64	454	552	736
1024	38	48	66	413	513	742
2048	36	45	60	362	475	600
4096	26	34	41	326	418	–
8196	18	22	37	314	348	–

Table 1. Speed-up of the CKY hardware approach over the CKY table algorithm.

References

1. A. V. Aho and J. D. Ullman. *The Theory of Parsing Translation and Compiling.* Prentice Hall, 1972.
2. Altera Corporation, APEX 20K Devices: System-on-a-Programmable-Chip Solutions, http://www.altera.com/products/devices/apex/apx-index.html.
3. Altera Corporation. Quartus II:system-on-a-programmable chip sowtfware. http://www.altera.com/products/software/quartus2/qts-index.html
4. J. Chang, O. Ibarra, and M. Palis. Parallel parsing on a one-way array of finite-state machines. *IEEE Transactions on Computers*, C-36(1):64–75, 1987.
5. E. Charniak. *Statistical Language Learning.* MIT Press, Cambridge, Massachusetts, 1993.
6. C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier. An FPGA-based coprocessor for the parsing of context-free grammars. In *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
7. C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier. An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, pages 590–594, 2001.
8. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90:61–79, 1991.
9. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.
10. N. D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
11. S. R. Kosaraju. Speed of recognition of context-free languages by array automata. *SIAM J. on Computers*, 4:331–340, 1975.
12. J. C. Martin. *Introduction to languages and the theory of computation (2nd Edition).* Mac-Graw Hill, 1996.
13. Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjölander, R. C. Underwood, and D. Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Research*, 22:5112–5120, 1994.
14. M. P. van Lohuizen. Survey on parallel context-free parsing techniques. Technical Report IMPACT-NLI-1997-1, Delft University of Technology, 1997.