PAPER Special Section on Parallel and Distributed Computing and Networking

An Efficient GPU Implementation of CKY Parsing Using the Bitwise Parallel Bulk Computation Technique*

Toru FUJITA[†], ***Student Member***, **Koji NAKANO**[†a)], **Yasuaki ITO**[†], ***and*** **Daisuke TAKAFUJI**[†], ***Members***

SUMMARY The main contribution of this paper is to present an efficient GPU implementation of bulk computation of the CKY parsing for a context-free grammar, which determines if a context-free grammar derives each of a lot of input strings. The bulk computation is to execute the same algorithm for a lot of inputs in turn or at the same time. The CKY parsing is to determine if a context-free grammar derives a given string. We show that the bulk computation of the CKY parsing can be implemented in the GPU efficiently using Bitwise Parallel Bulk Computation (BPBC) technique. We also show the rule minimization technique and the dynamic scheduling method for further acceleration of the CKY parsing on the GPU. The experimental results using NVIDIA TITAN X GPU show that our implementation of the bitwise-parallel CKY parsing for strings of length 32 takes 395μs per string with 131072 production rules for 512 non-terminal symbols.

key words: *parallel algorithms, bulk computation, bitwise operations, context-free grammar*

1. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [2]. 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 [2], [3]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [5], since they have hundreds of processor cores and very high memory bandwidth.

In our previous paper [1], we have introduced *the Bitwise Parallel Bulk Computation (BPBC) technique* to accelerate the computation. The BPBC technique supports ultimate fine grained bit parallelism and thus can achieve very high acceleration over the straightforward sequential computation. The BPBC technique simulates a combinational logic circuit for a lot of instances at the same time using the bitwise logical operations. More formally, let g be a function computed by a combinational logic circuit and $X_0, X_1, \ldots, X_{M-1}$ be the M inputs. By the BPBC technique $g(X_0), g(X_1), \ldots, g(X_{M-1})$ can be computed very efficiently. The idea of the BPBC technique is

- to store a bit of each input instance in a particular bit of words of data, say 32-bit integers, and
- to simulate the combinational logic circuit for 32 input vectors at the same time by bitwise logic operations supported by computing devices such as CPUs and GPUs.

We will show that the CKY parsing of a context-free grammar [6] for many inputs can be done very efficiently using the BPBC technique. 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 finite production rules, and S $(\in N)$ is the start symbol. Let $f(G, x)$ be a function such that G is a context-free grammar, $x = x_1 x_2 \cdots x_n$ is a string of length n, and $f(G, x)$ returns a Boolean value. Function $f(G, x)$ returns TRUE if and only if G derives x. It is well-known that the *CKY (Cocke-Kasami-Younger) parsing* [7] computes $f(G, x)$ in $O(n^3)$ time, where n is the length of x. The idea of the CKY parsing is to compute a 2-dimensional table $T[i, j]$ called CKY table by the dynamic programming technique. Each element of $T[i, j]$ stores a subset of non-terminal symbols that can derive substring $x_i x_{i+1} \cdots x_j$ by repeatedly applying production rules in P. Usually, each element of $T[i, j]$ is implemented as an array of size $|N|$ to maintain the subset of N. More specifically, $T[i, j][k] = 1$ if the k-th non-terminal symbol in N can derive substring $x_i x_{i+1} \cdots x_j$ by applying production rules. In most implementations of the CKY parsing, the value of each $T[i, j][k]$ is stored in a word, such as an 8-bit character or a 32-bit integer. Since each $T[i, j][k]$ stores 1-bit Boolean value, it is inefficient to use an array of words to store $T[i, j]$. Our idea to apply the BPBC technique to the CKY parsing is to compute 32 CKY tables for 32 input strings at the same time. Suppose that 32 input strings are given and we want to perform the CKY parsing for each of them. Let $T_0, T_1, \ldots, T_{31}$ denote 32 CKY tables to be computed. We store 32 values $T_0[i, j][k], T_1[i, j][k], \ldots, T_{31}[i, j][k]$ in a 32-bit integer for each i, j, and k. The CKY parsing can be done by iterative simulation of a combinational logic circuit [8], [9], the BPBC technique can be applied to it.

Further, we will present the minimization technique for reducing the number of gates of a combinational logic cir-

Manuscript received January 6, 2017.
Manuscript revised May 26, 2017.
Manuscript publicized August 4, 2017.

[†]The authors are with the Department of Information Engineering, Hiroshima University, Higashi-Hiroshima-shi, 739–8527 Japan.

*The preliminary version of this paper has been presented at the International Parallel and Distributed Processing Symposium Workshops (APDCM 2016) [1].

a) E-mail: nakano@cs.hiroshima-u.ac.jp

DOI: 10.1587/transinf.2017PAP0018

cuit of the CKY parsing. By reducing the number of gates, we can further accelerate the CKY parsing. In addition, we will introduce two scheduling methods, *the static scheduling* and *the dynamic scheduling* to assign input strings to CUDA blocks on the GPU. Our implementation results show that the dynamic scheduling is more efficient than the static scheduling if the number of CUDA blocks that we can launch is limited.

The parsing of context-free languages has many applications in various areas including natural language processing [10], [11], compiler construction [7], informatics [12], among others. Several studies have been devoted for accelerating the parsing of context-free languages [11], [13]–[15]. It has been shown that parsing of a string of length n can be done in $O((\log n)^2)$ time using n^6 processors on the PRAM [14]. 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 [15]. Later in [13], an algorithm that runs on a systolic array with n^2 finite-state processors with one-way communication running in linear time has been presented. In [16], it was shown that parsing can be accomplished on a one-way linear array of n^2 finite-state processors in linear time. Since these parallel algorithms need at least n processors, they are unrealistic for large n. Ciressan *et al.* [17], [18] and Bordim *et al.* [8], [9] have presented hardwares for the CKY parsing for context-free grammars and have tested them using FPGAs. In [8], it has been shown that the CKY parsing with 64 non-terminal symbols and 8192 production rules can be done in 162μs for an input string of length 32 using an APEX20K family FPGA. Because the circuit can run in about 35MHz, by estimating that the performance of the latest FPGA is about 10 times higher than the previous one, we can expect that the same circuit implemented in the latest FPGA may run in approximately 350MHz. Our GPU implementation can perform the same task only in 1.97μs. Hence our implementation is still more than 8 times faster than the hardware implementation even if the circuit is implemented on the latest FPGA and runs in 350MHz. Quite recently, GPU implementations of the CKY parsing have been presented [19], [20]. However, these implementation uses the straightforward bottom-up process, which performs only one CKY parsing. On the other hand, for a large number of inputs, the CKY parsing is performed in bioinformatics [21] and decision support systems [22]. In [21], to find RNA secondary structures from database, the CKY parsing is used. Also, in [22], the CKY parsing is used as preprocessing to parse a financial text stream consisting of millions of words. After that, by checking the sentiment of contexts whether it is positive or negative, the trend of the stock market is analyzed. The proposed CKY parsing with the BPBC technique can be performed if the size of 32 instances computed by each thread is the same. Our GPU implementation can be used for the above applications.

In the preliminary version of this paper [1], we have introduced the BPBC technique for the CKY parsing and its GPU implementation. Besides, in this paper, we show the rule minimization technique and the dynamic scheduling method for further acceleration of the CKY parsing on the GPU. The experimental results using NVIDIA TITAN X GPU show that our implementation of the bitwise-parallel CKY parsing for strings of length 32 takes 395μs per string with 131072 production rules for 512 non-terminal symbols.

This paper is organized as follows: In Sect. 2, we briefly explain the CKY parsing for context-free grammars and evaluate the performance. In Sect. 3, we show how we apply the BPBC technique to the CKY parsing and evaluate the performance. In Sect. 4, we explain the minimization technique for further acceleration of the CKY parsing. Section 5 shows the GPU implementation and the dynamic scheduling method. In Sect. 6, we show experimental results on the performance of the BPBC technique for the CKY parsing. Section 7 provides concluding remarks.

2. The CKY Parsing

The main purpose of this section is to briefly describe the CKY parsing and evaluate the performance.

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 finite production rules from N to $(N \cup \Sigma)^*$, and S $(\in N)$ is the start symbol. A context-free grammar is said to be in *Chomsky Normal Form* (CNF), if every production rule in P is in either form $A \to BC$ (*binary rule*) or $A \to a$ (*unary rule*), where A, B, and C are non-terminal symbols and a is a terminal symbol. Note that any context-free grammar can be converted into an equivalent CNF context-free grammar. For later reference, let p_2 and p_1 denote the numbers of binary and unary production rules, respectively.

We are interested in the parsing problem for a context-free grammar in CNF. More specifically, for a given CNF context-free grammar G and a string x over Σ, the parsing problem is a problem to determine if the start symbol S derives x by applying production rules in P. For example, let $G_{\text{example}} = (N, \Sigma, P, S)$ be a context-free grammar such that $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, and $P = \{S \to AB, S \to BA, S \to SS, A \to AB, B \to BA, A \to a, B \to b\}$. The context-free 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 $(1 \le i \le n)$ is in Σ. Let $T[i, j]$ $(1 \le i \le j \le n)$ denote a subset of N such that every A in $T[i, j]$ derives a substring $x_i x_{i+1} \cdots x_j$. The idea of the CKY parsing is to compute every $T[i, j]$ using the following relations:

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

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

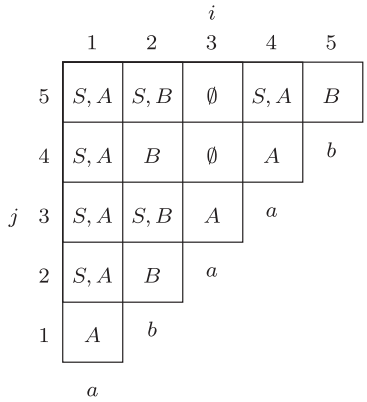

Fig. 1 The CKY table for G_{example} and *abaab*.

$$C \in T[k+1, j]\}$$

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

[CKY parsing]
1. $T[i, i] \leftarrow \{A \mid (A \rightarrow x_i) \in P\}$ for every i $(1 \leq i \leq n)$
2. $T[i, j] \leftarrow \emptyset$ for every i and j $(1 \leq i < j \leq n)$
3. for $j \leftarrow 2$ to n do
4. for $i \leftarrow j-1$ downto 1 do
5. for $k \leftarrow i$ to $j-1$ do
6. $T[i, j] \leftarrow T[i, j] \bigcup (T[i, k] \otimes_G T[k+1, j])$

The first two lines initialize the CKY table, and the next four lines compute the CKY table. Figure 1 illustrates the CKY table for G_{example} and the string *abaab*. Since $S \in T[1, 5]$, 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

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

times.

Let us evaluate the computing time t necessary to perform line 6, i.e., necessary to evaluate the binary operator $\otimes_G$. A traditional software approach (i.e, sequential algorithm), checks whether $B \in U$ and $C \in V$ for every production rule $A \rightarrow BC$ 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 $O(p_2)$ time. Thus, using the above approach, the CKY parsing can be done in $O(n^3 p_2)$ time.

Lemma 1: The CKY parsing for an input string of length n takes $O(n^3 p_2)$ time, where p_2 is the number of binary rules.

3. Bitwise Parallel Bulk Computation for CKY Parsing

This section is devoted to show how we apply the BPBC technique to the CKY parsing.

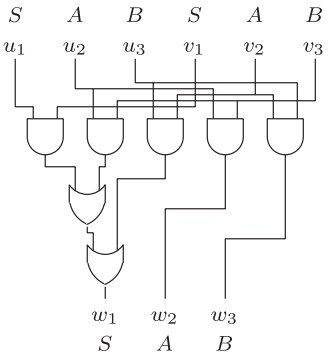

Fig. 2 The circuit for computing $\otimes_{G_{\text{example}}}$.

```
unsigned int u1,u2,u3,v1,v2,v3,w1,w2,w3;
w1 = (u2 & v3) | (u3 & v2) | (u1 & v1);
w2 = u2 & v3;
w3 = u3 & v2;
```

Fig. 3 A pseudocode for computing $\otimes_{G_{\text{example}}}$ in Fig. 2.

Suppose that a CNF context-free grammar $G = (N, \Sigma, P, S)$ is given. Let $N = \{N_1, N_2, \ldots, N_b\}$ be a set of non-terminal symbols, where b is the number of non-terminal symbols. Recall that the CKY parsing repeatedly computes $U \otimes_G V = \{A \mid (A \rightarrow BC) \in P, B \in U, \text{ and } C \in V\}$. We will show that computation of $U \otimes_G V$ can be represented by a combinational logic circuit. Let U and V $(\in 2^N)$ be represented by b-bit binary vectors $u_1 u_2 \cdots u_b$ and $v_1 v_2 \cdots v_b$, respectively, such that $u_i = 1$ iff $N_i \in U$. Also, let $U \otimes_G V = w_1 w_2 \cdots w_b$. For a particular w_k $(1 \leq k \leq b)$, we are going to show how w_k is computed. Let $N_k \rightarrow N_{i_1} N_{j_1}$, $N_k \rightarrow N_{i_2} N_{j_2}$, ..., and, $N_k \rightarrow N_{i_s} N_{j_s}$ be the production rules in P whose non-terminal symbol in the left-hand side is N_k. Clearly, we can compute w_k by the following formula:

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

This formula corresponds to a combinational circuit with s AND gates and $s-1$ OR gates and, the value of w_k can be computed by simulating the circuit. Figure 2 illustrates a circuit for G_{example} in Sect. 2. Clearly, the combinational circuit for $\otimes_G$ has p_2 AND gates and less than p_2 OR gates. Figure 3 shows a pseudocode for computing $\otimes_{G_{\text{example}}}$ in Fig. 2.

Since the computation of $\otimes_G$ can be done by simulating a combinational logic circuit, we can use the BPBC technique for the CKY parsing, which repeatedly computes $\otimes_G$. We assume that M input strings $X_0, X_1, \ldots, X_{M-1}$ of length n each are given. Our goal is to determine if $G = (N, \Sigma, P, S)$ can generate X_i for all i $(0 \leq i \leq M-1)$ by the CKY parsing. We assume that a CPU used for the CKY parsing operates on d-bit words. We partition the input strings into $\frac{M}{d}$ groups of d strings each. Let $x_{i,j}$ denote the j-th character of X_i. We show how we determine if G can generate X_i for the first group. We use $|N|$ d-bit integers to represent subsets of non-terminal symbols N for d input strings of the first group. Each bit of d-bit integers corresponds to one of the d input

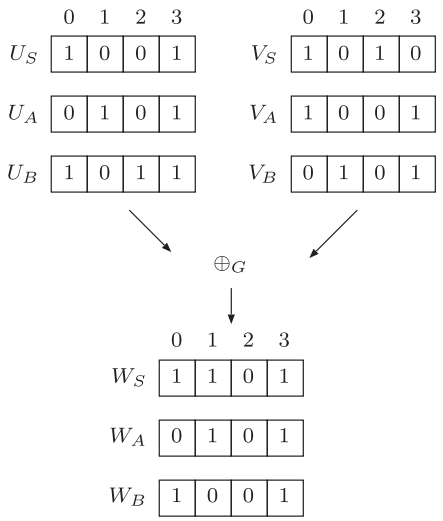

Fig. 4 The computation of $\otimes_G$ for four instances.

strings. Using these integers, we can compute $\otimes_G$ by the bitwise operations very efficiently. Figure 4 illustrates the computation of $\otimes_G$ for an example of a context-free grammar shown in Sect. 2. It uses three 4-bit integers to represent subsets of non-terminal symbols. Each of the 4 bits correspond to the following computation in terms of $\otimes_G$:

0: $\{S, B\} \otimes_G \{S, A\} \to \{S, B\}$
1: $\{A\} \otimes_G \{B\} \to \{S, A\}$
2: $\{B\} \otimes_G \{S\} \to \{\}$
3: $\{S, A, B\} \otimes_G \{A, B\} \to \{S, A, B\}$

Since the computation of $\otimes_G$ can be represented by a combinational logic circuit, we can compute $\otimes_G$ for d pairs of inputs by bitwise logic operations. For example, the computation illustrated in Fig. 4 can be done by bitwise logic operations as follows:

$$W_S \leftarrow (U_A \wedge V_B) \vee (U_B \wedge V_A) \vee (U_S \wedge V_S)$$
$$W_A \leftarrow U_A \wedge V_B$$
$$W_B \leftarrow U_B \wedge V_A$$

Using this idea, we can perform the CKY parsing shown in Sect. 2. We perform the CKY parsing for d input strings at the same time. The computation of $\otimes_G$ performed in line 6 of the CKY parsing can be done by $O(p_2)$ bitwise logic operations. Hence, the CKY parsing for d input strings can be done in $O(n^3 p_2)$ time. Since we have $\frac{M}{d}$ groups, we have

Theorem 2: The CKY parsing of M input strings of length n each can be done in $O(\frac{Mn^3 p_2}{d})$ time by the BPBC technique.

From Lemma 1, the CKY parsing for M input strings of length n can be done in $O(Mn^3 p_2)$. Thus, the BPBC technique can accelerate it by a speed-up factor of d.

Suppose that the BPBC technique is used on a parallel machine with P processor cores. Clearly, it can perform the CKY parsing for dP input strings at the same time. Thus, we have,

Corollary 3: The CKY parsing of M input strings of length n each can be done in $O(\frac{Mn^3 p_2}{dP})$ time by the BPBC technique on a parallel machine with P processor cores.

From Lemma 1, the CKY parsing of M input strings takes $O(Mn^3 p_2)$ time on a single CPU. Hence, we can say that a parallel machine with P processor cores may accelerate the CKY parsing by a speed up factor of up to $O(dP)$. For example, NVIDIA TITAN X utilized in our experiments has $P = 3584$ cores and each word has $d = 32$ bits. Thus, the BPBC technique on the GPU can achieve a speed up factor of up to $3584 \times 32 = 114688$ over the conventional word-wise computation on a single processor core. Of course, this speedup factor is just a theoretical upper bound and actual speedup factor is much smaller, because a GPU processor core has less computing power than a processor core of a CPU and the GPU has larger overhead than the CPU due to large memory access latency, limited memory bandwidth and smaller clock performance, among others.

4. Minimization Technique for CKY Parsing

The main purpose of this section is to show the minimization technique for the combinational logic circuit of the CKY parsing. Since the computation cost for the CKY parsing by the BPBC technique is proportional to the number of gates, we can accelerate the CKY parsing if the size of the corresponding logic circuit is reduced.

Suppose that a grammar $G = (N, \Sigma, P, S)$ is given. We assume that the production rules whose non-terminal symbol in the left-hand side is S are $\{S \leftarrow SS, S \leftarrow SB, S \leftarrow AA, S \leftarrow AB, S \leftarrow BS, S \leftarrow BB\}$. As we have mentioned in Sect. 3, we can compute w_S by a combinational logic circuit as follows:

$$w_S \leftarrow (u_S \wedge v_S) \vee (u_S \wedge v_B) \vee (u_A \wedge v_A) \vee (u_A \wedge v_B) \vee (u_B \wedge v_S) \vee (u_B \wedge v_B)$$

In this formula, we can compute w_S using 11 logic gates. On the other hand, we can see that both the first rule $(u_S \wedge v_S)$ and the second rule $(u_S \wedge v_B)$ have u_S. Hence, we can join these rules as follows:

$$(u_S \wedge v_S) \vee (u_S \wedge v_B)$$
$$\Downarrow$$
$$u_S \wedge (v_S \vee v_B)$$

By joining the rules, we can reduce the number of gates from 11 to 10. Using this idea, we minimize the combinational logic circuit of the rules. Our minimization technique consists of two steps, *factoring step* and *simulated annealing step*.

4.1 Factoring Step

We create a matrix for each non-terminal symbol as illustrated in Fig. 5. Each element of the matrix (u_i, v_j) is 1 if

$$w_S \to (u_S \wedge v_S) \vee (u_S \wedge v_B) \vee (u_A \wedge v_A) \vee (u_A \wedge v_B) \vee (u_B \wedge v_S) \vee (u_B \wedge v_B)$$

	v_S	v_A	v_B	
u_S	1	0	1	2
u_A	0	1	1	2
u_B	1	0	1	2
	2	1	3	

Fig. 5 The production rule matrix of w_S.

	v_S	v_A	v_B			v_S	v_A	v_B	
u_S	1	0	1	2	u_S	1	0	0	1
u_A	0	1	1	2	u_A	0	1	0	1
u_B	1	0	1	2	u_B	1	0	0	1
	2	1	3			2	1	0	

Fig. 6 Updating the production rules matrix.

$(u_i \wedge v_j)$ is included in the combinational logic circuit of P, and 0 if it is not included for all i, j ($\in N$). After creating the matrix, we count the number of 1's in each row and each column as shown in Fig. 5. We repeatedly select a column/row with the maximum number of 1's and join the rules. For example, since a column corresponding to v_B has the maximum number of 1's, rules for it are joined as follows:

$$\begin{aligned} w_S &\leftarrow (u_S \wedge v_S) \vee (u_S \wedge v_B) \vee (u_A \wedge v_A) \\ &\vee\ (u_A \wedge v_B) \vee (u_B \wedge v_S) \vee (u_B \wedge v_B) \\ &\Downarrow \\ w_S &\leftarrow ((u_S \vee u_A \vee u_B) \wedge v_B) \\ &\vee\ (u_S \wedge v_S) \vee (u_A \wedge v_A) \vee (u_B \wedge v_S) \end{aligned}$$

After joining the rules, all values in the column set to 0 as illustrated in Fig. 6. We repeat the same joining procedure until all elements of the matrix becomes 0.

Using this technique for the matrix in Fig. 5, we finally obtain 3 joined rules as follows:

$$\begin{aligned} w_S &\leftarrow ((u_S \vee u_A \vee u_B) \wedge v_B) \\ &\vee\ ((u_S \vee u_B) \wedge v_S) \\ &\vee\ (u_A \wedge v_A) \end{aligned}$$

From these joined rules, we can see that the number of gates is reduced from 11 to 8.

4.2 Simulated Annealing Step

We will show that we can further reduce the number of gates by the simulated annealing technique using an example. In Sect. 4.1, both the joined rules $((u_S \vee u_A \vee u_B) \wedge v_B)$ and $((u_S \vee u_B) \wedge v_S)$ have $(u_S \vee u_B)$. Hence, we can join these 2 rules as follows:

$$\begin{aligned} &((u_S \vee u_A \vee u_B) \wedge v_B) \ \vee\ ((u_S \vee u_B) \wedge v_S) \\ &\Downarrow \\ &((u_S \vee u_B) \wedge (v_S \vee v_B)) \ \vee\ (u_A \wedge v_B) \end{aligned}$$

Since the joined rule $((u_S \vee u_A \vee u_B) \wedge v_B)$ has u_A, we must add rule $(u_A \wedge v_B)$. Thus, the number of gates can be reduced by 1. However, in general, we may need to add more rules, and the number of gates may increase.

We use the simulated annealing technique [23], [24] for reducing the number of gates. First, we select two rules at random from the joined rules by the factoring technique and evaluate the number of gates that can be reduced if we join them. If the number of gates is reduced, we actually join them. It makes no sense to join them if the number of gates increases. However, since we use the simulated annealing technique, we actually join them with some small probability even if the number of gates increases. We repeat this procedure until we are not able to find better formula in enough many trials.

5. GPU Implementation

This section shows the GPU implementation of the CKY parsing using the BPBC technique.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. The shared memory is an extremely fast on-chip memory with capacity of 16–96 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, its capacity of 1.5–12 Gbytes is large, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access [5], [25]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

We use the global memory of the GPU to store the CKY tables since the size of the shared memory is not large enough to store the tables in it. The CKY parsing computes elements of the CKY table in the order illustrated in Fig. 7. The elements are computed from the bottom row. In each row, they are computed from right to left. Hence, we use the local memory of CUDA to cache the value of a row currently computed. Note that the local memory may be allocated in registers in the streaming multiprocessor if small, and in the off-chip DRAM if large. Hence, it makes sense to use the local memory to cache a row. Also, since the capacity of the local memory is limited, it is not possible to store all elements of the CKY table in it.

Each thread performs the CKY parsing for 32 input strings at the same time using bitwise operations for 32-bit integers. We arrange 32 threads for each CUDA block, a group of threads of CUDA. Since each CUDA block performs the CKY parsing for a subset of $32 \cdot 32 = 1024$ input strings, $\frac{M}{1024}$ CUDA blocks are invoked for M input strings.

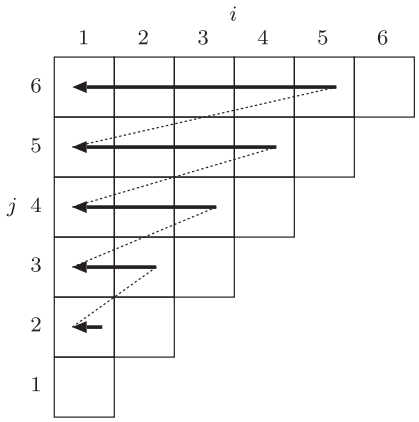

Fig. 7 The computation order of the CKY table.

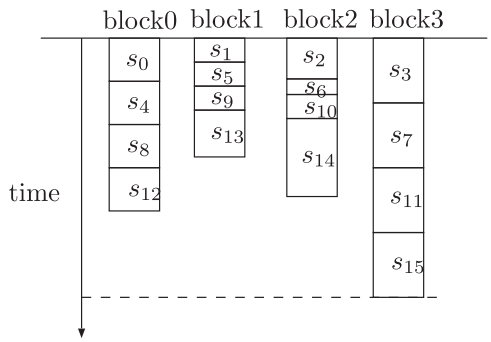

Fig. 8 The timeline of the static scheduling.

However, if M is quite large, it is not possible to launch $\frac{M}{1024}$ CUDA blocks, because the total size of the CKY tables for them exceeds that of the global memory of the GPU. Since the number of CKY tables that can be stored in the global memory is limited, it makes sense to invoke fewer CUDA blocks each of which performs the CKY parsing for multiple subsets of 1024 input strings each. In other words, the space for the CKY tables in the global memory is reused.

We will show two scheduling methods to assign multiple subsets to each CUDA block, *the static scheduling* and *the dynamic scheduling*. In the static scheduling method, each block simply performs the CKY parsing for the same number of subsets. Since the computing time for the CKY parsing may vary, CUDA blocks do not terminate at the same time. In Fig. 8, 4 CUDA blocks are invoked for the CKY parsing of subsets s_0, s_1, ..., and s_{15}. In the static scheduling, each block performs the CKY parsing for fixed $\frac{16}{4} = 4$ subsets and block 3 runs much longer than the others.

On the other hand, in the dynamic scheduling method, each subset is dynamically assigned to each CUDA block. In this method, each CUDA block is assigned to a subset for the CKY parsing one by one. After a CUDA block completes the CKY parsing for the assigned subset, a new subset is assigned to it as illustrated in Fig. 9. To implement the dynamic scheduling, we use *atomic add instruction* supported by CUDA for a global variable c initialized to 0. When CUDA blocks are launched, the first thread of each block

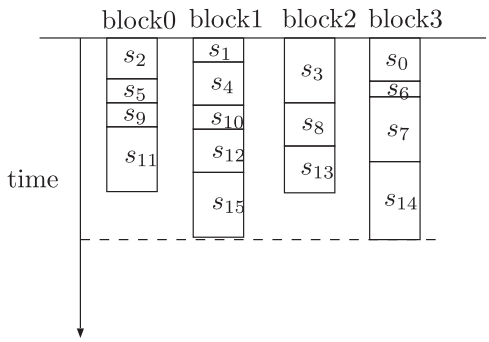

Fig. 9 The timeline of the dynamic scheduling.

executes atomicAdd(c,1), which exclusively adds 1 to c and returns the value of c before adding. A CUDA block performs the CKY parsing for a subset s_i, where i is the return value of atomicAdd(c,1). Each CUDA block repeats the same procedure until the CKY parsing for all subsets is completed. If the return value is larger than the number of subsets, it terminates. Since atomicAdd(c,1) returns $0, 1, 2, \ldots$ in each call, the CKY parsing for every subset is done one by one properly. Using this technique, a CUDA block which takes a lot of time for the CKY parsing is assigned a new subset later. Thus, the running time of CUDA blocks is equalized so that they terminate almost at the same time, and the computing time for completing all CKY parsing is shortened.

The CKY parsing by the dynamic scheduling runs faster than that by the static scheduling. In particular, if all CUDA blocks are allocated to streaming multiprocessors at the same time and they run in parallel, then the dynamic scheduling approach runs much faster. On the other hand, if a kernel invokes more CUDA blocks and they are allocated to streaming multiprocessors in turn, the advantage of the CKY parsing by the dynamic scheduling is small. The reason is as follows. In CUDA, the number of CUDA blocks that can reside and concurrently run on a streaming multiprocessor is limited. It depends on the numbers of threads and CUDA blocks, the size of the registers and shared memory used by each CUDA block. If all CUDA blocks reside and run on streaming multiprocessors, the running time of them varies and the worst one determines the running time of the CKY parsing as illustrated in Fig. 8. Suppose that a kernel invokes more CUDA blocks than the total number of CUDA blocks that can run on all streaming multiprocessors at the same time. CUDA blocks that cannot be allocated to a streaming multiprocessor wait for termination of execution of running CUDA blocks. When a CUDA block running on a streaming multiprocessor terminates, one of waiting CUDA blocks is allocated to it. This scheduling is done by the CUDA block scheduler. We can think that the CUDA block scheduler performs the CUDA-blockwise dynamic scheduling, which equalizes the running time of streaming multiprocessors. In other words, the static scheduling for the CKY parsing automatically performs the CUDA-blockwise dynamic scheduling. Our dynamic scheduling is

finer-grained than the CUDA-blockwise dynamic scheduling in the sense that the running times of CUDA blocks are equalized in subsetwise for each subset of 1024 input strings. Hence, we can expect the dynamic scheduling runs faster than the static scheduling for this case, but the difference of the performance is not large. The experimental results in Sect. 6 show the correctess of this observation.

6. Experimental Results

This section shows experimental results using Intel Core i7-6700K (4.0GHz) CPU and NVIDIA TITAN X (1.4GHz) GPU. NVIDIA TITAN X has 28 streaming multiprocessors with 128 cores each. Hence, it has totally $28 \times 128 = 3584$ processor cores. We use the BPBC technique both for the CPU implementation and the GPU implementation. Although Intel Core i7-6700K has 4 processor cores, we have used only one processor core to evaluate sequential algorithm. We may accelerate the computation by a speedup factor of up to 4 if we implement a parallel algorithm that uses all 4 processor cores. Since our goal is not to compare the computing powers of Intel Core i7-6700K and NVIDIA TITAN X, we have not implemented a 4-parallel algorithm on Intel Core i7-6700K.

Table 1 shows the number of gates of the combinational logic circuit of the CKY parsing. The experiment is performed for 32, 64, 128, 256, and 512 non-terminal symbols and $32, 64, \ldots, 131072$ binary production rules. Note that the number p_2 of binary production rules must be $|N| \leq p_2 < |N|^3$, where $|N|$ is the number of non-terminal symbols. If $|N| > p_2$, there exists a non-terminal symbol that are not in the left-hand side of a binary production rule. Since a binary production rule in form $A \rightarrow BC$ includes 3 non-terminal symbols, it make no sense to have $|N|^3$ or more distinct binary rules. Thus, the table does not include the experiment for values p_2 out of this range. We have selected p_2 production rules from $|N|^3$ distinct rules at random. In each non-terminals, the first column *original* and the second column *min* represent the number of gates for an original logic circuit and that obtained after minimization shown in Sect. 4. The third column *ratio* represents the ratio of *min* to *original*. We can see that the number of gates can be reduced for almost pairs of non-terminals and rules. The value of ratio is decreased as the number of rules increases in each non-terminals. In other words, we can reduce the number of gates more if the number of rules is larger. On the other hand, the ratio is increased as the number of non-terminal symbols increases in each rules, because the num-

Table 1 The number of gates of the combinational logic circuit of the CKY parsing and the ratio of min to original.

	32 non-terminals			64 non-terminals			128 non-terminals			256 non-terminals			512 non-terminals		
p_2	original	min	ratio	original	min	ratio	original	min	ratio	original	min	ratio	original	min	ratio
32	64	63	0.98	-	-	-	-	-	-	-	-	-	-	-	-
64	128	124	0.97	128	128	1.00	-	-	-	-	-	-	-	-	-
128	256	245	0.96	256	254	0.99	256	254	0.99	-	-	-	-	-	-
256	512	466	0.91	512	496	0.97	512	507	0.99	512	511	1.00	-	-	-
512	1024	864	0.84	1024	963	0.94	1024	1008	0.98	1024	1020	1.00	1024	1023	1.00
1024	2048	1582	0.77	2048	1854	0.91	2048	1986	0.97	2048	2028	0.99	2048	2042	1.00
2048	4096	2802	0.68	4096	3469	0.85	4096	3886	0.95	4096	4031	0.98	4096	4076	1.00
4096	8192	4722	0.58	8192	6318	0.77	8192	7437	0.91	8192	7939	0.97	8192	8119	0.99
8192	16384	7436	0.45	16384	11300	0.69	16384	13883	0.85	16384	15471	0.94	16384	16134	0.98
16384	32768	11834	0.36	32768	19662	0.60	32768	25338	0.77	32768	29674	0.91	32768	31841	0.97
32768	-	-	-	65536	32924	0.50	65536	45420	0.69	65536	55452	0.85	65536	61973	0.95
65536	-	-	-	131072	56097	0.43	131072	80471	0.61	131072	101236	0.77	131072	118600	0.90
131072	-	-	-	262144	101484	0.39	262144	140777	0.54	262144	182008	0.69	262144	221668	0.85

Table 2 The running time (in seconds) of the bitwise-parallel CKY parsing on the GPU for 4194304 strings of length 32 each.

	32 non-terminals 2048 blocks			64 non-terminals 1024 blocks			128 non-terminals 512 blocks			256 non-terminals 256 blocks			512 non-terminals 128 blocks		
p_2	static	dynamic	spd-up	static	dynamic	spd-up	static	dynamic	spd-up	static	dynamic	spd-up	static	dynamic	spd-up
32	0.794	0.806	0.985	-	-	-	-	-	-	-	-	-	-	-	-
64	1.08	1.11	0.975	1.46	1.51	0.967	-	-	-	-	-	-	-	-	-
128	1.18	1.19	0.990	1.95	1.99	0.977	2.75	2.86	0.964	-	-	-	-	-	-
256	1.12	1.14	0.979	2.39	2.29	1.05	3.75	3.76	1.00	6.14	5.48	1.12	-	-	-
512	1.11	1.12	0.991	2.14	2.15	0.995	5.30	5.13	1.03	11.9	10.8	1.10	30.6	16.2	1.89
1024	1.21	1.20	1.01	2.22	2.22	1.00	7.66	7.64	1.00	19.7	17.8	1.10	49.0	24.5	2.00
2048	1.28	1.27	1.01	2.53	2.53	1.00	8.50	8.11	1.05	26.4	23.5	1.13	96.0	35.9	2.67
4096	1.32	1.26	1.05	5.40	5.16	1.05	10.4	9.17	1.13	41.5	36.9	1.12	173	65.0	2.67
8192	3.60	3.46	1.04	8.32	7.92	1.05	15.9	14.3	1.12	72.0	63.8	1.13	332	114	2.92
16384	6.58	6.49	1.01	10.9	10.1	1.08	25.9	22.5	1.15	136	122	1.11	647	222	2.91
32768	-	-	-	16.7	15.2	1.10	42.7	36.3	1.18	263	236	1.11	1310	414	3.16
65536	-	-	-	34.5	31.8	1.09	72.3	60.5	1.19	484	461	1.05	2600	819	3.17
131072	-	-	-	60.7	55.6	1.09	125	104	1.21	930	847	1.10	5890	1660	3.56

Table 3 The running time (per string in microseconds) of the bitwise-parallel CKY parsing for strings of length 32.

	32 non-terminals			64 non-terminals			128 non-terminals			256 non-terminals			512 non-terminals		
p_2	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up
32	3.25	0.278	11.7	-	-	-	-	-	-	-	-	-	-	-	-
64	5.62	0.350	16.1	5.97	0.448	13.3	-	-	-	-	-	-	-	-	-
128	10.3	0.369	27.8	10.5	0.565	18.7	14.7	0.761	19.3	-	-	-	-	-	-
256	18.9	0.356	53.2	19.8	0.632	31.3	26.3	0.985	26.7	28.9	1.40	20.7	-	-	-
512	33.6	0.352	95.6	39.1	0.601	65.0	51.9	1.31	39.7	55.4	2.66	20.8	60.9	3.95	15.4
1024	57.2	0.376	152	74.1	0.617	120	104	1.91	54.6	106	4.34	24.4	119	5.92	20.2
2048	95.6	0.387	247	136	0.690	197	199	2.02	98.4	210	5.68	36.9	216	8.65	25.0
4096	167	0.385	434	234	1.32	178	375	2.27	165	401	8.89	45.1	429	15.6	27.5
8192	244	0.914	267	411	1.97	209	635	3.49	182	774	15.3	50.6	811	27.2	29.8
16384	339	1.63	208	745	2.48	300	1170	5.44	215	1430	29.2	49.1	1630	53.1	30.7
32768	-	-	-	1230	3.70	333	2340	8.73	268	2520	56.4	44.6	2900	98.7	29.4
65536	-	-	-	1900	7.66	248	3950	14.5	272	4660	110	42.3	5380	195	27.6
131072	-	-	-	3720	13.4	278	6210	24.8	251	9160	202	45.4	10900	395	27.7

ber of rules that can be joined is decreased. From the table, using the minimization technique, the number of gates can be reduced by 18% on average.

Table 2 shows the running time of the CKY parsing by our BPBC technique for 4194304 strings of length 32 with the static scheduling and the dynamic scheduling on the GPU. We use the minimized circuit in Table 1 for each implementations. Also, we use 32-bit unsigned integers and arrange 32 threads for each CUDA block. Thus, a CUDA block performs the CKY parsing for $32 \cdot 32 = 1024$ input strings in parallel. Thus, we partition 4194304 strings into $\frac{4194304}{1024} = 4096$ subsets and the CKY parsing for each subset is performed by a CUDA block. From the capacity of the global memory of NVIDIA TITAN X, 2048, 1024, 512, 256, and 128 CUDA blocks are invoked for 32, 64, 128, 256, and 512 non-terminals, respectively, which occupies 8 Gbytes in the global memory of the GPU for the CKY tables.

From Table 2, we can see that the dynamic scheduling runs much faster than the static scheduling if the number of non-terminals is large. For example, the dynamic scheduling can run more than 1.89–3.56 times faster for 512 non-terminals. However, the running times are not so different for smaller number of non-terminals. This is because all CUDA blocks reside and run on streaming multiprocessors at the same time for 512 non-terminals. From the CUDA profiler executed for our CKY parsing by the BPBC technique, each streaming multiprocessor can execute 8 CUDA blocks. Thus, at most $28 \times 8 = 224$ CUDA blocks run at the same time on 28 streaming multiprocessors in NVIDIA TITAN X. Since a kernel invokes 128 CUDA blocks for 512 non-terminals, all CUDA blocks run at the same time and the dynamic scheduling runs much faster than the static scheduling. On the other hand, 256 CUDA blocks are invoked for 256 non-terminals and so $256-224 = 32$ CUDA blocks are not dispatched to streaming multiprocessors when a kernel is called. Thus, since the CUDA-blockwise dynamic scheduling works for the CKY parsing by the static scheduling, the dynamic scheduling runs a little faster than the static scheduling.

Table 3 shows the running time per string of the CKY parsing for strings of length 32 on the CPU and the GPU. We use the minimized circuit in Table 1 both for the CPU and the GPU implementations. Also, we use the dynamic scheduling method for GPU implementations. Clearly, the running time of the CPU implementation is almost proportional to the number of gates of the minimized circuit. For the same number of binary production rules, the CPU implementation for more non-terminal symbols takes more time. However, the running time is sublinear to the number of non-terminal symbols, although it must be linear from the theoretical point of view. This is because the locality of memory access. If the context-free grammar has fewer non-terminal symbols, then each of them are accessed more frequently and the memory cache mechanism works more efficiently.

Similarly, the GPU implementation also takes more time if the context-free grammar has more non-terminal symbols. In addition to the locality of memory access, fewer active threads increase the running time. For example, if the context-free grammar has 32 non-terminal symbols, we can launch $2048 \cdot 32 = 65536$ threads. On the other hand, if the context-free grammar has 512 non-terminal symbols, we can launch only $128 \cdot 32 = 576$ threads. In general, to maximize the memory access bandwidth, more threads must be invoked at the same time. Hence, the running time per input string is rather increased because fewer CKY tables are computed using fewer threads. From Table 3, the GPU implementation is 434 times faster than the CPU implementation when the context-free grammar has 32 non-terminal symbols and 4096 binary production rules.

7. Concluding Remarks

In this paper, we have presented an efficient GPU implementation of the bulk computation of the CKY parsing. We have used the Bitwise Parallel Bulk Computation (BPBC) technique for accelerating the bulk computation. Also, we have presented the minimization technique for reducing the number of gates of the combinational logic circuit of the CKY parsing. In addition, we have used the dynamic scheduling method for efficiently assignment of subsets of input strings to CUDA blocks. The experimental results using

NVIDIA TITAN X GPU show that our implementation of the bitwise-parallel CKY parsing for strings of length 32 takes 395μs per string with 131072 production rules for 512 non-terminals.

References

[1] T. Fujita, K. Nakano, and Y. Ito, "Bitwise parallel bulk computation on the GPU, with application to the CKY parsing for context-free grammars," Proc. International Parallel and Distributed Processing Symposium Workshops, pp.589–598, May 2016.
[2] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.
[3] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," Proc. International Conference on Networking and Computing, pp.279–280, Nov. 2010.
[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," March 2015.
[5] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," International Journal of Networking and Computing, vol.1, no.2, pp.260–276, July 2011.
[6] J.C. Martin, Introduction to languages and the theory of computation (2nd Edition), MacGraw-Hill, 1996.
[7] A.V. Aho and J.D. Ullman, The Theory of Parsing Translation and Compiling, Prentice Hall, 1972.
[8] J.L. Bordim, Y. Ito, and K. Nakano, "Accelarating the CKY parsing using FPGAs," IEICE Trans. Inf. & Syst., vol.E86-D, no.5, pp.811–818, May 2003.
[9] J.L. Bordim, O.H. Ibarra, Y. Ito, and K. Nakano, "Instance-specific solutions to accelerate the CKY parsing for large contex-free grammars," International Journal on Foundations of Computer Science, vol.15, no.2, pp.403–416, April 2014.
[10] E. Charniak, Statistical Language Learning, MIT Press, Cambridge, Massachusetts, 1993.
[11] M.P. van Lohuizen, "Survey on parallel context-free parsing techniques," Tech. Rep. IMPACT-NLI-1997-1, Delft University of Technology, 1997.
[12] 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, vol.22, no.23, pp.5112–5120, 1994.
[13] J.H. Chang, O.H. Ibarra, and M.A. Palis, "Parallel parsing on a one-way array of finite-state machines," IEEE Trans. Comput., vol.C-36, no.1, pp.64–75, 1987.
[14] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.
[15] S.R. Kosaraju, "Speed of recognition of context-free languages by array automata," SIAM J. Computers, vol.4, no.3, pp.331–340, 1975.
[16] O.H. Ibarra, T. Jiang, and H. Wang, "Parallel parsing on a one-way linear array of finite-state machines," Theoretical Computer Science, vol.85, no.1, pp.53–74, Aug. 1991.
[17] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An FPGA-based coprocessor for the parsing of context-free grammars," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.
[18] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars," Proc. International Conference on Field Programmable Logic and Applications (FPL), pp.590–594, 2001.
[19] Y. Yi, C.Y. Lai, S. Petrov, and K. Keutzer, "Efficient parallel CKY parsing on GPUs," Proc. International Conference on Parsing Technologies, pp.175–185, 2011.
[20] K.-H. Kim, S.-M. Choi, H. Lee, K.L. Man, and Y.-S. Han, "Parallel CYK membership test on GPUs," Proc. International Conference on Network and Parallel Computing (LNCS 8707), pp.157–168, Sept. 2014.
[21] R.D. Dowell and S.R. Eddy, "Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction," BMC Bioinformatics, vol.5, no.1, p.71, 2004.
[22] S.W.K. Chan and M.W.C. Chong, "Sentiment analysis in financial texts," Decision Support Systems, vol.94, pp.53–64, 2017.
[23] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, "Optimization by simulated annealing," Science, vol.220, no.4598, pp.671–680, May 1983.
[24] V. Černý, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," Journal of Optimization Theory and Applications, vol.45, no.1, pp.41–51, Jan. 1985.
[25] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

Toru Fujita received the B.E., M.E. and D.E. degrees from Hiroshima University in 2014, 2015 and 2017, respectively.

Koji Nakano received the B.E., M.E. 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 JAIST. He has been a full professor at School of Engineering, Hiroshima University from 2003.

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.

Daisuke Takafuji received B.E. and M.E. degrees from Hiroshima University in 1993 and 1995 respectively. He is currently an assistant professor of the Graduate School of Engineering, Hiroshima University. His research interests include design and analysis of algorithms for graphs and parallel computation.