Bitwise Parallel Bulk Computation on the GPU, with Application to the CKY Parsing for Context-free Grammars

Toru Fujita, Koji Nakano, Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The main contribution of this paper is to present Bitwise Parallel Bulk Computation (BPBC) technique, to accelerate bulk computation, which executes the same algorithm for a lot of instances in turn or in parallel. The idea of the BPBC technique is to simulate a combinational logic circuit for 32 inputs at the same time using bitwise logic operators for 32-bit integers supported by most processing devices. We will show that the BPBC technique works very efficiently on a CPU as well as on a GPU. As a simple example of the BPBC, we first show that the pairwise sums of a lot of integers can be computed faster using the BPBC technique, if the values of input integers are not large. We also show that the CKY parsing for context-free grammars can be implemented in the GPU efficiently using the BPBC technique. The experimental results using Intel Core i7 CPU and GeForce GTX TITAN X GPU show that the GPU implementation for the CKY parsing can be more than 400 times faster than the CPU implementation.**

***Keywords*-Parallel algorithms, bulk computation, bitwise operations, context-free grammar**

I. Introduction

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. 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 [1], [4], [5], [6]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [7], 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 [8], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [7]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12Gbytes, 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 bank conflict* of the shared memory access and *the coalescing* of the global memory access [5], [8], [9]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed one by one. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. 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.

A sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input [10]. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i] + b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent of the values stored in b. *The bulk execution* of a sequential algorithm is to execute it for many different inputs in turn or at the same time. For example, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each. We can compute the prefix-sums of each b_j $(0 \leq j \leq p-1)$ by executing the prefix-sum algorithm on a single CPU in turn or on a parallel computer in parallel. The bulk execution has many applications. For example, the conventional FFT algorithm [11] for n points running in $O(n \log n)$ time is oblivious. In practical signal processing, an input stream is equally partitioned into many blocks, and the FFT algorithm is executed for each block in turn or in parallel. This is exactly the bulk execution of the FFT algorithm. In our previous paper [10], we have introduced the bulk execution of sequential algorithms, and show that they can be implemented in the GPU very efficiently if they are oblivious. In [12], we have also developed a conversion tool C2CU, which automatically generates an efficient CUDA program for the bulk computation from an oblivious sequential C language program. Quite surprisingly, the bulk computation can achieve speed-up factor of more than 100, even if it does not use the shared memory of the GPU.

The main contribution of this paper is to present a novel technique, *the Bitwise Parallel Bulk Computation (BPBC) technique*. The bulk execution for oblivious algorithms pre-

sented in our previous papers [10], [12] is *word-wise* in the sense that each of data is stored in a word such as a 32-bit integer. On the other hand, the BPBC technique supports ultimate fine grained bitwise 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. More formally, let f 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 $f(X_0)$, $f(X_1)$, ..., $f(X_{M-1})$ can be computed very efficiently. The idea of the BPBC technique is

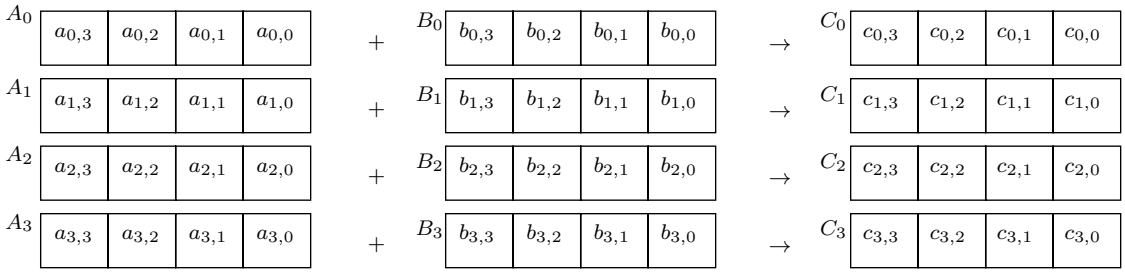
- to store data bits of each input instance in a particular bit of words of data, say 32-bit integers, and
- to simulate the combinational logic circuit for 32 input vectors at the same time by bitwise logic operations supported by computing devises such as CPUs and GPUs.

The BPBC technique is inspired by our previous work [13], which shows an efficient simulation of the Conway's Game of Life [14], a well-known cellular automaton. The next state of each cell in a cellular automaton is computed by simulating a combinational logic circuit. Thus, a state of each cell is stored in a bit of a 32-bit integer, and the combinational logic circuit to compute the next state is simulated by bitwise logic operations. One of the contributions of this paper is to generalize this idea and to name it the BPBC technique.

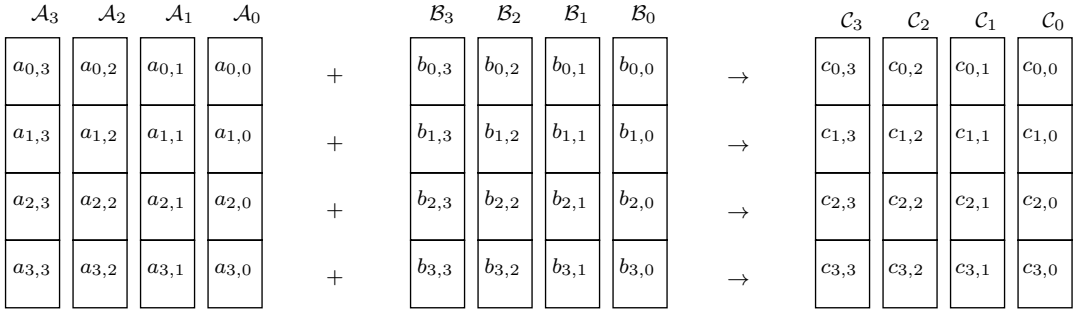
As a very simple example of the BPBC, we first show that the pairwise sums of integers can be accelerated if the value of integers are small. Suppose that 32 pairs (A_0, B_0), (A_1, B_1), ..., (A_{31}, B_{31}) of 32-bit integers are given, and we want to compute the pairwise sums A_0+B_0, A_1+B_1, ..., $A_{31}+B_{31}$. Clearly, 32 addition operations can complete the computation of pairwise sums. The BPBC technique for this task simulates thirty two 32-bit ripple-carry adders by 32-bit bitwise operations, such as bitwise OR, AND, and XOR. Figure 1 illustrates the naive pairwise addition and the BPBC pairwise addition for four pairs of 4-bit numbers. The naive pairwise addition performs addition operation four times. On the other hand, the BPBC pairwise addition simulates a 4-bit ripple-carry adder for $(\mathcal{A}_3, \mathcal{A}_2, \mathcal{A}_1, \mathcal{A}_0)$ and $(\mathcal{B}_3, \mathcal{B}_2, \mathcal{B}_1, \mathcal{B}_0)$ to obtain $(\mathcal{C}_3, \mathcal{C}_2, \mathcal{C}_1, \mathcal{C}_0)$. In most practical application programs, A's and B's do not have full 32 bits. In some applications, all integers stored in 32-bit integer may have only 10-20 bits. The running time of a naive pairwise addition cannot be reduced even if the these numbers are very small. On the other hand, the computation of pairwise sums by the BPBC can be accelerated if values of A's and B's are small. For example, if these numbers are less than 2^{16}, then we can compute the pairwise sums by simulating 16-bit ripple-carry adders. Our experimental results show that, the pairwise sum by the BPBC is faster than the naive pairwise addition if the number of bits of input numbers is at most 19 on the CPU, that is, the values are less than 2^{19}. Also, the pairwise sum by the BPBC is faster than the naive pairwise addition if the number of bits of input numbers is at most 29 on the GPU, that is, the values are less than 2^{29}. Also, the GPU implementation of the pairwise sums by the bitwise parallel bulk computation can be more than 20 times faster than the CPU implementation. Thus, we can say that the BPBC technique is a potent method to accelerate the bulk computation using the GPU.

As a practical and complicated application, we will show that the CKY parsing of a context-free grammar [15] 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, $x = x_0 x_1 \cdots x_{n-1}$ 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* [16] computes $f(G, x)$ in $O(n^3)$ time, where n is the length of x [16]. 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 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 combinational logic circuit [17], [18], the BPBC technique can be applied to it.

The parsing of context-free languages has many applications in various areas including natural language processing [19], [20], compiler construction [16], informatics [21], among others. Several studies have been devoted for accelerating the parsing of context-free languages [20], [22], [23], [24]. 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 [23]. 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 [24]. Later in [22], 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 [25], 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.* [26], [27] and Bordim *et al.* [17], [18] have presented hardwares for the CKY parsing for context-free grammars and have tested them using FPGAs. In [17], it



(1) naive pairwise addition for computing $A_i + B_i \rightarrow C_i$ for $i = 0, 1, 2, 3$



(2) BPBC pairwise addition

Fig. 1. The naive pairwise addition and the BPBC pairwise addition for four pairs of 4-bit numbers

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. Our GPU implementation can do the same task in only 3.68μs per one input string. Hence our implementation is more than 40 times faster than the FPGA implementation. Quite recently, GPU implementations of CKY parsing have been presented [28], [29]. However, these implementation uses the straightforward bottom-up process, which performs only one CKY parsing.

This paper is organized as follows: In Section II, we introduce the Bitwise Parallel Bulk Computation (BPBC) technique that simulates a combinational logic circuit for a lot of instances, and show theoretical analysis of performance of the BPBC. Section III briefly explains the CKY parsing for context free grammars and evaluates the performance. In Section IV, we show how we apply the BPBC technique to the CKY parsing and evaluate the performance. In Section V, we explain the Unified Memory Machine (UMM) [30], [31], which is a theoretical model of GPU computing using the global memory, and evaluate the performance of the BPBC on the UMM. Section VI shows experimental results on the performance of the BPBC technique for the pairwise sum computation and the CKY parsing. Section VII provides concluding remarks and future work.

II. Bitwise Parallel Bulk Computation (BPBC)

The main purpose of this section is to show the idea of Bitwise Parallel Bulk Computation (BPBC). This idea is work well not only for a multi-core machine but also for a single CPU.

Let $f : \{0,1\}^m \rightarrow \{0,1\}^n$ be a function with m input bits and n output bits. Since f is a function, there exists a combinational logic circuit that computes f. Let $X_0, X_1, \ldots, X_{d-1}$ be d inputs of m bits each. Suppose that we want to compute $f(X_0), f(X_1), \ldots, f(X_{d-1})$. We can evaluate these values one by one using a single CPU. Also, we can use d processor cores and compute $f(X_i)$ for each X_i $(0 \leq i \leq d-1)$ using one processor each. Our new idea, Bitwise Parallel Bulk Computation (BPBC) can perform this computation much faster than these straightforward sequential and parallel algorithms simulating the combinational logic circuit independently for all inputs.

Let $x_{i,0}x_{i,1}\cdots x_{i,m-1}$ denotes m bits of each X_i $(0 \leq i \leq d-1)$. Further, let $x_{0,j}x_{1,j}\cdots x_{d-1,j}$ be $\mathcal{X}_j$. We assume that CPU can handle d-bit word and each $\mathcal{X}_j$ is stored in a d-bit word. By bitwise logic operations for $\mathcal{X}_j$, we can simulate a combinational logic circuit for computing f, and can obtain the values of $f(X_0), f(X_1), \ldots, f(X_{d-1})$ at the same time. For example, let $f(a,b,c) = (y,z)$ such that $y = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$ and $z = a \oplus b \oplus c$. In other words, f is a function simulating a full adder. Also, let $a_ib_ic_i$ denotes three bits of each X_i $(0 \leq i \leq d-1)$. We assume that we have three d-bit words $A = a_0a_1\cdots a_{d-1}$, $B = b_0b_1\cdots b_{d-1}$, and $C = c_0c_1\cdots c_{d-1}$. We want to compute $Y = y_0y_1\cdots y_{d-1}$ and $Z = z_0z_1\cdots z_{d-1}$ such that $(y_i, z_i) = f(a_i, b_i, c_i)$ for all i $(0 \leq i \leq d-1)$. Clearly, by performing

$$\begin{aligned} Y &\leftarrow (A \wedge B) \vee (B \wedge C) \vee (C \wedge A), \text{ and} \\ Z &\leftarrow A \oplus B \oplus C, \end{aligned}$$

where $\wedge$, $\vee$, and $\oplus$ represent bitwise AND, OR, and XOR operators in this context. After executing this program, Y and

Z store the resulting values of $f(A, B, C)$.

Let us see a bit more complicated example. Suppose that we have two sequences of d numbers with d bits each. Let $A = A_0, A_1, \ldots, A_{d-1}$ and $B = B_0, B_1, \ldots, B_{d-1}$ denote these two sequences, and $a_{i,j}$ and $b_{i,j}$ be the j-th bits of A_i and B_i, respectively. Our goal is to compute sequence $C_0, C_1, \ldots, C_{d-1}$ of d numbers with d bits each, such that $C_i = (A_i + B_i) \bmod 2^d$ for each i $(0 \leq i \leq d-1)$. Clearly, we can obtain C by computing the pairwise sums of two sequences A and B in $O(d)$ time by an obvious sequential algorithm as illustrated in Figure 1 (1). Let us apply the BPBC technique for this problem. Let $\mathcal{A}_j$ and $\mathcal{B}_j$ be words of d bits each such that $\mathcal{A}_j = a_{0,j}a_{1,j}\cdots a_{d-1,j}$ and $\mathcal{B}_j = b_{0,j}b_{1,j}\cdots b_{d-1,j}$. The goal is to compute the sum $\mathcal{C}_j = c_{0,j}c_{1,j}\cdots c_{d-1,j}$ such that $c_{i,j}$ is the j-th bit of C_i as illustrated in Figure 1 (2). We can obtain $\mathcal{C}$ by simulating a d-bit ripple-carry adder as follows:

$$
\begin{array}{rcl}
\mathcal{C}_0 & \leftarrow & \mathcal{A}_0 \oplus \mathcal{B}_0 \\
\mathcal{T} & \leftarrow & \mathcal{A}_0 \wedge \mathcal{B}_0 \\
\mathcal{C}_1 & \leftarrow & \mathcal{A}_1 \oplus \mathcal{B}_1 \oplus \mathcal{T} \\
\mathcal{T} & \leftarrow & (\mathcal{A}_1 \wedge \mathcal{B}_1) \vee (\mathcal{A}_1 \wedge \mathcal{T}) \vee (\mathcal{B}_1 \wedge \mathcal{T}) \\
\mathcal{C}_2 & \leftarrow & \mathcal{A}_2 \oplus \mathcal{B}_2 \oplus \mathcal{T} \\
\mathcal{T} & \leftarrow & (\mathcal{A}_2 \wedge \mathcal{B}_2) \vee (\mathcal{A}_2 \wedge \mathcal{T}) \vee (\mathcal{B}_2 \wedge \mathcal{T}) \\
 & \vdots & \\
\mathcal{C}_{d-1} & \leftarrow & \mathcal{A}_{d-2} \oplus \mathcal{B}_{d-2} \oplus \mathcal{T}
\end{array}
$$

In this algorithm, $\mathcal{T}$ is a d-bit temporal variable used to store carry bits. Clearly, this algorithm runs $O(d)$ time. Hence, the computing time is the same as the obvious sequential algorithm. However, if the numbers stored in A and B are small, we can omit the computation. More specifically, suppose that all numbers in A and B are less than K. Since the pairwise sums in C are less than $2K$, all $\mathcal{C}_i$'s $(\log K + 1 \leq i \leq d-1)$ are zero. Thus, we can omit the computation for these $\mathcal{C}_i$'s and the computing time can be decreased to $O(\log K)$.

Let us evaluate the computation performed the BPBC technique. Again, let $f : \{0,1\}^m \rightarrow \{0,1\}^n$ be a function with m input bits and n output bits. We assume that f can be computed by a combinational logic circuit with s gates. We have M inputs $X_0, X_1, \ldots, X_{M-1}$ with m bits each and compute $f(X_i)$ for all i $(0 \leq i \leq M-1)$. Since $f(X_i)$ can be computed in $O(s)$ time by simulating the combinational logic circuit, we can evaluate all $f(X_i)$ for all i in $O(sM)$ time.

To apply our bitwise computation technique, we partition the input into $\frac{M}{d}$ groups of d inputs each. Let $x_{i,j}$ denote the j-th bit of X_i as before. Since the first group has d inputs $X_0, X_1, \ldots, X_{d-1}$, We store each j-th bit $(0 \leq j \leq d-1)$ in a d-bit integer $\mathcal{X}_j$ such that $\mathcal{X}_j = x_{0,j}x_{1,j}\cdots x_{d-1,j}$. Clearly, the values of $f(X_i)$ for all i $(0 \leq i \leq d-1)$ can be computed in $O(s)$ time by simulating the combinational logic circuit by bitwise logic operations. The values of $f(X_i)$'s remaining groups can be computed in the same way. Thus, we have,

Theorem 1: The bulk computation of simulating a combinational circuit with s gates for M inputs can be done in $O(\frac{sM}{d})$ time using a single d-bit processor.

III. 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 $A \rightarrow BC$ (*binary rule*) or $A \rightarrow a$ (*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 $P = \{S \rightarrow AB, S \rightarrow BA, S \rightarrow SS, A \rightarrow AB, B \rightarrow BA, A \rightarrow a, B \rightarrow b\}$. 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 $x = x_1x_2\cdots x_n$ be a string of length n, where each x_i $(1 \leq i \leq n)$ is in Σ. Let $T[i, j]$ $(1 \leq i \leq j \leq n)$ denote a subset of N such that every A in $T[i, j]$ derives a substring $x_ix_{i+1}\cdots x_j$. The idea of the CKY parsing is to compute every $T[i, j]$ using the following relations:

$$
\begin{array}{rcl}
T[i,i] & = & \{A \mid (A \rightarrow x_i) \in P\} \\
T[i,j] & = & \bigcup_{k=i}^{j-1} \{A \mid (A \rightarrow BC) \in P, B \in T[i,k], \text{ and } \\
 & & \quad C \in T[k+1,j]\}
\end{array}
$$

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] ← {A | (A → x_i) ∈ P} for every i (1 ≤ i ≤ n)
2. T[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.             T[i,j] ← T[i,j] ⋃ (T[i,k] ⊗_G T[k+1,j])
```

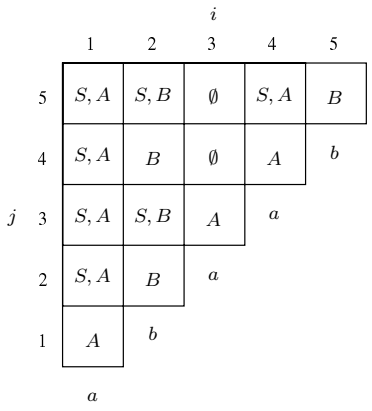


Fig. 2. The CKY table for G_{example} and $abaab$.

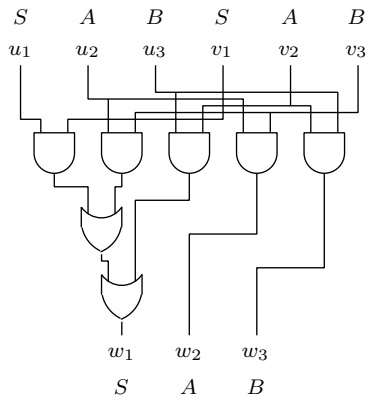


Fig. 3. The circuit for computing $\otimes_{G_{\text{example}}}$.

The first two lines initialize the CKY table, and the next four lines compute the CKY table. Figure 2 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, the running time is

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

Let us evaluate the computing time t necessary to perform line 6, i.e., necessary to evaluate the binary operator $\otimes_G$. A straightforward sequential algorithm checks 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 $O(p_2)$ time. Thus, using the above approach, the CKY parsing can be done in $O(n^3 p_2)$ time.

Lemma 2: 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.

IV. Bitwise Parallel Bulk Computation for CKY Parsing

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

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 $U \otimes_G V = \{A \mid (A \to BC) \in P, B \in U, \text{ and } C \in V\}$. 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 ($1 \le k \le b$), we are going to show how w_k is computed. Let $N_k \to N_{i_1} N_{j_1}$, $N_k \to N_{i_2} N_{j_2}$, ..., and, $N_k \to N_{i_s} N_{j_s}$ be the binary 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 3 illustrates a circuit for G_{example} in Section III. Clearly, the combinational circuit for $\otimes_G$ has p_2 AND gates and less than p_2 OR gates.

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 generates X_i for all i ($0 \le i \le M-1$) by the CKY parsing. Similarly, we partition the input strings into $\frac{M}{d}$ groups of d strings each, because we use a d-bit CPU. 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 strings. Using these integers, we can compute $\otimes_G$ by the bitwise operation very efficiently. Figure 4 illustrates the computation of $\otimes_G$ for an example of a context-free grammar shown in Section III. 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 input by bitwise logic operations. For example, the computation illustrated Figure 4 can be done by bitwise logic operations as follows:

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

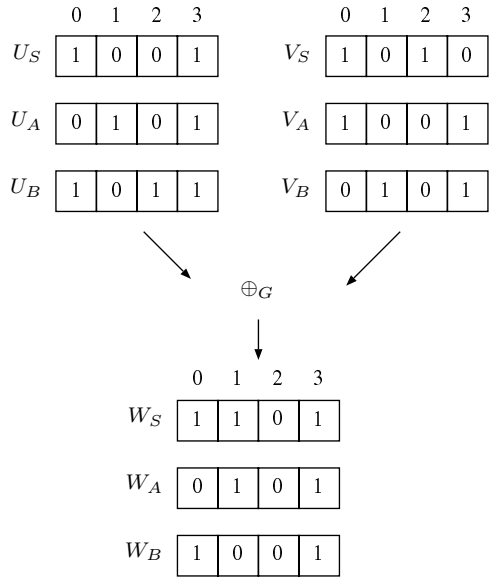


Fig. 4. The computation of $\otimes_G$ for four instances

Using this idea, we can perform the CKY parsing shown in Section III. 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 3: The CKY parsing of M input strings of length n each can be done in $O(\frac{Mn^3p_2}{d})$ time by the BPBC technique.

From Lemma 2, the CKY parsing for M input strings of length n can be done in $O(Mn^3p_2)$. Thus, the BPBC technique can accelerate it by a speed-up factor of d.

V. The BPBC on the UMM and performance analysis

The main purpose of this section is to show that the BPBC can be implemented in the GPU very efficiently. For this purpose, we first introduce the Unified Memory Machine (UMM) [30], [31], which captures the essence of global memory access of the GPU. The reader should refer to [30], [31] for the details of the the UMM. We evaluate the performance of algorithms using the BPBC technique on the UMM to show that they are efficient from the theoretical point of view. Since the BPBC technique implemented in the GPU does not use the shared memory, the UMM can be used for the theoretical analysis of the performance.

Let us define the UMM with width w and latency l. Let $m[i]$ ($i \geq 0$) denote the memory cell with address i. We assume that each memory cell can store d-bit integer. The memory of the UMM is partitioned into address groups $A[0], A[1], \ldots$ such that each $A[j]$ ($j \geq 0$) stores $m[j \cdot w]$, $m[j \cdot w + 1]$, $\ldots$, $m[(j+1) \cdot w - 1]$. The reader should refer Figure 5 that illustrates the memory for $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in the figure. Let p be the number of threads of the UMM and $T(0), T(1), \ldots, T(p-1)$ be the p threads. We assume that p is a multiple of w. The p threads are partitioned into $\frac{p}{w}$ groups called *warps* with w threads each. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1) \cdot w - 1)\}$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory in the same time. More specifically, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w thread in $W(i)$ sends memory access requests, one request per thread, to the memory banks.

For the memory access, each warp sends memory access requests to the memory through the l-stage pipeline registers. We assume that each stage can store the memory access requests destined for the same address group. For example, since the memory access requests by $W(0)$ are separated in three address groups in the figure, they occupy three stages of the pipeline registers. Also, those by $W(1)$ are in the same address group, they occupy only one stage. In general, if memory access requests by a warp are destined for k address groups, they occupy k stages. We assume that the memory access is completed as soon as the request is dequeued from the pipeline. Thus, all memory access requests by $W(0)$ and $W(1)$ in the figure are completed in 3(address groups)$+1$(address group)$+5$(latency)$-1 = 8$ time units. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

Let us implement the BPBC technique in the UMM and evaluate the performance. As before, we have M inputs $X_0, X_1, \ldots, X_{M-1}$ with m bits each and compute $f(X_i)$ for all i ($0 \leq i \leq M-1$), where f is a function computed by a combinational logic circuit with s gates. For this purpose, we partition the input into $\frac{M}{d}$ groups of d inputs each and each thread i ($0 \leq i \leq \frac{M}{d} - 1$) on the UMM computes the value of i-th group by simulating the combinational logic circuit. For example, thread 0 computes the values of $f(X_0), f(X_1), \ldots, f(X_{d-1})$. Each thread also uses s words to store the output of s gates, and thus, it totally uses $m + s$ words. We arrange $m + s$ words used by $\frac{M}{d}$ threads in a 2-dimensional array of $(m+s) \times \frac{M}{d}$ d-bit words as illustrated in Figure 6. Each thread accesses to $m+s$ words in a column and simulates the combinational logic circuit for d inputs. We call this arrangement *the column-wise arrangement*. Since words in the same row are allocated in consecutive addresses, memory access to words in the same row by w threads in a warp occupies pipeline registers in one stage. Thus, memory access requests by $\frac{M}{d}$ threads to the same row occupies $\frac{M}{wd}$ stages and the memory access is completed in at most $O(\frac{M}{wd}+l)$ time units. Since every thread issues $O(s)$ memory access requests to simulate the combinational logic circuit, we have,

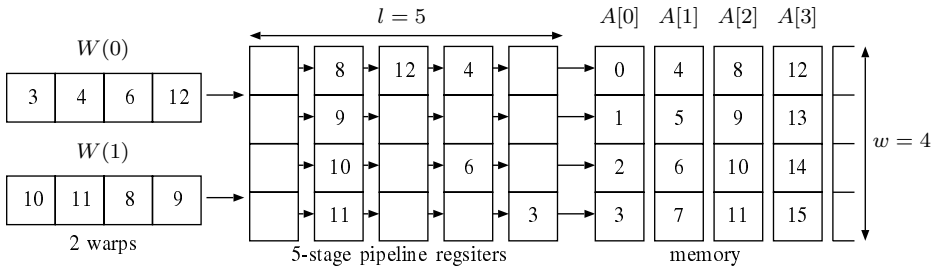


Fig. 5. The UMM with width $w = 4$ and latency $L = 5$

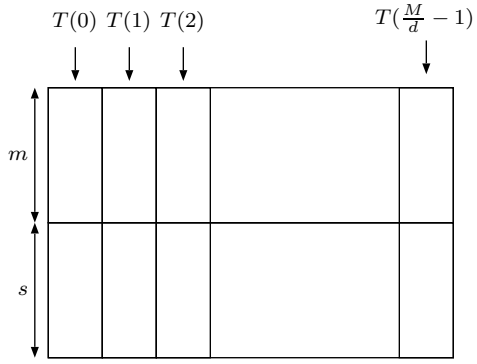


Fig. 6. The column-wise arrangement of $m + s$ words on the memory

Theorem 4: The BPBC represented by a combinational logic circuit with s gates for M inputs can be done in $O(\frac{sM}{wd} + sl)$ time units using $\frac{M}{d}$ threads on the UMM with d-bit words, width w and latency l.

Clearly, at most wd bits in the memory of the UMM can be accessed in a time unit. To simulate a combinational logic circuit with s gates for M inputs, at least sM memory requests are necessary. Thus, this task takes at least $\Omega(\frac{sM}{wd})$ time on the UMM and it is time optimal if $sl \leq \frac{sM}{wd}$, that is, if $M \geq wdl$ holds.

Next, let us evaluate the time for pairwise sum of M pairs of two numbers using theorem 4. We assume that all numbers are less than K. As we have discussed, the combinational logic circuit to compute the pairwise sums has $O(\log K)$ gates. Thus, from Theorem 4, we have,

Corollary 5: The pairwise sums of M pairs of two numbers less than K can be computed in $O(\frac{M \log K}{wd} + l \log K)$ time units using $\frac{M}{d}$ threads on the UMM with d-bit words, width w and latency l.

Similarly, this computation is optimal if $M \geq wdl$.

The reader should have no difficulty to implement the CKY parsing by the BPBC technique shown for Lemma 3 on the UMM. Let M be the number of input strings of length n. We use $\frac{M}{d}$ threads on the UMM to perform the CKY parsing. Recall that the CKY parsing for a context-free grammar G repeatedly computes the function $\otimes_G$, which can be represented by a combinational logic circuit with $O(p_2)$ gates. Let b be the number of non-terminal symbols. Each thread computes d CKY tables at the same time. Since each CKY table has $O(n^2)$ entries, each thread uses $O(bn^2)$ words to store all entries of d CKY tables. We arrange these $O(bn^2)$ words in a column of 2-dimensional array similarly to the column-wise arrangement shown in Figure 6. Using the column-wise arrangement, memory access is coalesced, and memory access requests by w threads in a warp occupies only one pipeline stage. Each thread performs $O(n^3 p_2)$ memory access operations and each memory access by $\frac{M}{d}$ threads can be done in $O(\frac{M}{wd} + l)$ time units. Thus, we have,

Corollary 6: The CKY parsing for M input strings of length n can be done in $O(\frac{n^3 p_2 M}{wd} + n^3 p_2 l)$ time units on the UMM d-bit words, width w and latency l.

VI. GPU IMPLEMENTATION AND EXPERIMENTAL RESULTS

The main purpose of this section is to show experimental results using Intel Core i7-4790 (3.6GHz) CPU and GeForce GTX TITAN X (1GHz) GPU. GeForce GTX TITAN X has 24 streaming multiprocessors with 128 cores each. Hence, it has totally $24 * 128 = 3072$ processor cores. Since we use the BPBC technique, we have not used the shared memory of streaming multiprocessors on the GPU. Although Intel Core i7-4790 has 4 processor cores, we have used only one processor core to evaluate sequential algorithms. 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 i7-4790 and GeForce GTX TITAN X, we have not implemented a 4-parallel algorithm on Intel Core i7-4790.

A. Pairwise summing by the BPBC technique

We have evaluated the running time for pairwise summing for $2^{20} = 1048576$ pairs of 32-bit unsigned integers stored in the global memory of the GPU. We assume that all numbers are less than 2^k and evaluated the performance for every k ($1 \leq k \leq 32$). The implementation of a sequential algorithm for naive pairwise summing is obvious. It simply computes the sum of two integers one by one. Bitwise pairwise summing is performed by simulating k-bit ripple-carry adder circuit for two k-bit integers. In GPU implementations, we invokes 1024 CUDA blocks with 32 threads each and each thread computes the sums of 32 pairs by the BPBC.

Table I shows the running time and the speed-up factors. The naive sequential algorithm runs about 0.676 ms, while GPU performs the same computation in 0.0566 ms. Thus, the GPU implementation is 11.9 times faster than the CPU for the naive algorithm. The running time of bitwise CPU/GPU

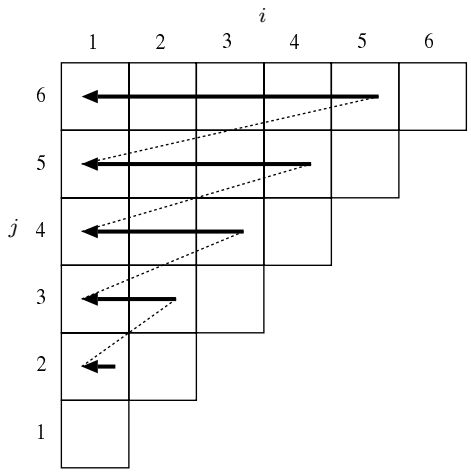


Fig. 7. The computation order of the CKY table

implementations is almost proportional to the value of k. Also, the bitwise implementations are faster if $k \leq 19$ on the CPU and $k \leq 29$ on the GPU. We can say that, the BPBC is a potent method if the upper bound of input numbers is limited, and the GPU acceleration works very effectively. For example, if input numbers is guaranteed to be in the range of 16-bit short integer, the pairwise sums for 1048576 pairs can be computed only in 0.0223ms by the BPBC technique. On the other hand, naive pairwise summing takes 0.0565ms. The sequential algorithm for the bitwise summing technique runs 0.562ms. Hence, the GPU bitwise pairwise summing is the best method among others.

B. CKY parsing by the BPBC technique

We first show the implementation of GPU implementation for CKY parsing by the BPBC technique. We use the global memory of the GPU to store the CKY tables. The CKY parsing computes elements of the CKY table in the order illustrated in Figure 7. The elements are computed from 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. Even if it is allocated in the off-chip DRAM, it may be accessed faster than the global memory, which is also arranged in the off-chip DRAM. This is because an element of the local memory is accessed by a particular thread, and cache mechanism may work efficiently for the local memory. 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.

Recall that the CKY parsing by the BPBC on the UMM uses $\frac{M}{d}$ threads for M input strings. Since we use 32-bit unsigned integers, $d = 32$ and $\frac{M}{32}$ threads are invoked. We arrange 32 threads for each CUDA block, a CUDA kernel for the CKY parsing invokes $\frac{M}{1024}$ CUDA blocks with 32 threads each.

Table II shows the running time of the bitwise CKY parsing for M input strings of length 32. The context-free grammar has 32/64 non-terminal symbols and 8192 production rules. Each thread uses a 32-bit unsigned register to store the current status of 32 non-terminal symbols. Note that, the total number of possible binary production rules is $32^3 = 32768$. We have selected 8192 production rules form these rules at random. We have evaluated the running time for $M = 1024$ to 2097512 ($= 2^{21}$). Clearly, the running time of the CKY parsing by the CPU is proportional to M, because it just repeats the CKY parsing for M input strings. On the other hand, the running time by the GPU is not. Recall that, from Corollary 6, the CKY parsing on the UMM takes $O(\frac{n^3 p_2 M}{wd} + n^3 p_2 l)$ time units. Roughly speaking, from Table II, we can think that $O(n^3 p_2 l) \approx 0.4$ for 32 non-terminal symbols. Also, $O(\frac{n^3 p_2 M}{wd}) \approx 3.85 - 0.4 = 3.45$ when $M = 2097152$. Hence, $O(\frac{n^3 p_2}{wd}) \approx \frac{3.45}{2097152} \approx 1.65 \cdot 10^{-6}$. Thus, we can say that, the latency overhead is 0.4 seconds and the throughput is 1.65 μs per CKY parsing. Further, M must be so large that $M \geq 262144$ to hide the latency overhead.

Table III shows the running time per string of the CKY parsing for strings of length 32 using the BPBC technique. The experiment is performed for 32, 64, 128, 256, and 512 non-terminal symbols and 32, 64, ..., 131072 binary production rules. Note that the number p_2 of binary production rules must be $|N| \leq p_2 \leq |N|^3$, 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 is not possible to have more than $|N|^3$ distinct binary rules. Thus, the table does not include the experiment for values p_2 out of this range. From the capacity of the global memory of GeForce GTX TITAN X, we can implement 2097152, 1048576, 524288, 262144, and 131072 CKY tables, respectively, which occupies 8Gbytes in the global memory of the GPU. Clearly, the running time of the CPU implementation is almost proportional to p_2, because the number of bitwise operations performed is $O(p_2)$. Also, for the same number of binary production rules, the CPU implementation for more non-terminal symbols takes more time. For example, the CKY parsing takes 1310 μs for 32 non-terminal symbols and 16384 binary rules, while it runs 1740 μs if the context-free grammar has 64 non-terminal symbols. This is because the locality of memory access. Roughly speaking, each non-terminal symbol is accessed three times for each binary rule. Hence, we can think that about $\frac{16384 \cdot 3}{32} = 1536$ memory access operations is performed for each of 32 non-terminal symbols. On the other hand, if the context-free grammar has 64 non-terminal symbols, each non-terminal symbols are accessed expected $\frac{16384 \cdot 3}{64} = 768$ times. If the context-free grammar has fewer non-terminal symbols, then each of them are accessed more frequently and the memory cache mechanism work more efficiently.

Similarly, the GPU implementation also takes more time if the context-free grammar has more non-terminal symbols. In

TABLE I
THE RUNNING TIME (IN MILLISECONDS) OF THE PAIRWISE SUMS FOR 1048576 PAIRS AND THE SPEED-UP FACTORS

k	naive		bitwise		speed-up factors			
	CPU	GPU	CPU	GPU	GPU over CPU		bitwise over naive	
					naive	bitwise	CPU	GPU
1	0.682	0.0564	0.0475	0.00119	12.1	39.8	14.4	47.2
2	0.676	0.0566	0.109	0.00230	11.9	47.6	6.18	24.6
3	0.676	0.0564	0.149	0.00337	12.0	44.1	4.55	16.7
4	0.677	0.0566	0.196	0.00445	12.0	44.1	3.45	12.7
5	0.676	0.0566	0.221	0.00487	11.9	45.4	3.06	11.6
6	0.676	0.0569	0.250	0.00601	11.9	41.6	2.70	9.46
7	0.676	0.0565	0.278	0.00703	12.0	39.5	2.44	8.04
8	0.676	0.0564	0.310	0.00814	12.0	38.1	2.18	6.93
9	0.676	0.0566	0.340	0.00863	11.9	39.4	1.99	6.56
10	0.674	0.0563	0.376	0.0119	12.0	31.7	1.79	4.75
11	0.676	0.0565	0.403	0.0134	12.0	30.2	1.68	4.23
12	0.675	0.0565	0.434	0.0171	12.0	25.4	1.56	3.31
13	0.675	0.0565	0.464	0.0176	12.0	26.4	1.45	3.22
14	0.676	0.0566	0.496	0.0199	11.9	24.9	1.36	2.85
15	0.676	0.0565	0.529	0.0209	12.0	25.3	1.28	2.70
16	0.680	0.0565	0.562	0.0223	12.0	25.2	1.21	2.54
17	0.676	0.0566	0.592	0.0224	12.0	26.4	1.14	2.52
18	0.676	0.0570	0.623	0.0241	11.9	25.8	1.08	2.36
19	0.676	0.0567	0.657	0.0271	11.9	24.2	1.03	2.09
20	0.676	0.0567	0.691	0.0289	11.9	23.9	0.977	1.96
21	0.676	0.0569	0.726	0.0327	11.9	22.2	0.930	1.74
22	0.676	0.0565	0.761	0.0359	12.0	21.2	0.889	1.57
23	0.676	0.0567	0.796	0.0387	11.9	20.6	0.849	1.47
24	0.677	0.0570	0.830	0.0410	11.9	20.3	0.815	1.39
25	0.676	0.0565	0.865	0.0440	12.0	19.6	0.782	1.28
26	0.676	0.0566	0.902	0.0476	12.0	19.0	0.750	1.19
27	0.675	0.0564	0.936	0.0494	12.0	19.0	0.721	1.14
28	0.676	0.0568	0.973	0.0519	11.9	18.7	0.695	1.09
29	0.676	0.0567	1.01	0.0544	11.9	18.5	0.671	1.04
30	0.676	0.0566	1.04	0.0570	11.9	18.3	0.649	0.993
31	0.676	0.0569	1.08	0.0598	11.9	18.0	0.627	0.951
32	0.675	0.0568	1.41	0.0616	11.9	22.9	0.480	0.923

TABLE II
THE RUNNING TIME (IN SECONDS) OF THE BITWISE-PARALLEL CKY PARSING FOR STRINGS OF LENGTH 32 AND CONTEXT FREE GRAMMARS WITH 32/64 NON-TERMINAL SYMBOLS AND 8192 PRODUCTION RULES AND THE SPEED-UP FACTOR

	32 non-terminals			64 non-terminals		
M	CPU	GPU	spd-up	CPU	GPU	spd-up
1024	0.679	0.412	1.65	0.637	0.432	1.47
2048	1.30	0.402	3.25	1.25	0.435	2.87
4096	2.60	0.405	6.40	2.43	0.437	5.56
8192	5.18	0.408	12.7	4.85	0.441	11.0
16384	10.4	0.387	26.8	9.71	0.423	22.9
32768	20.7	0.383	54	19.4	0.425	45.6
65536	41.4	0.399	104	38.7	0.459	84.4
131072	82.8	0.434	191	77.4	0.769	101
262144	166	0.754	220	155	0.989	157
524288	331	0.983	337	310	1.94	159
1048576	663	1.93	344	615	3.87	159
2097152	1330	3.85	344	-	-	-

addition to the locality of memory access, fewer active threads increases the running time. For example, if the context-free grammar has 32 non-terminal symbols, the global memory of the GPU can store 2097152 CKY tables, which are computed by $\frac{2097152}{32} = 65536$ threads. On the other hand, the global memory of the GPU can store 1048576 CKY tables, which are computed by $\frac{1048576}{32} = 32768$ 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 III, the GPU implementation is more than 400 times faster than the CPU implementation when the context-free grammar has 32 non-terminal symbols and 16384/32768 binary production rules. However, if it has many non-terminal symbols and the global memory of the GPU can store fewer CKY table, we cannot attain high speed-up factor.

VII. CONCLUDING REMARKS AND FUTURE WORK

In this paper, we have introduced the Bitwise Parallel Bulk Computation (BPBC) technique. We apply the BPBC technique to two computations, the pairwise sum computation and the CKY parsing. To show the BPBC technique works efficiently, we give theoretical analysis of performance of these two computations. We also implemented them on the GPU and showed that the GPU implementations can be more than 400 times faster than the CPU implementation.

The BPBC technique is a potent method if we need to execute a same algorithm for a lot of input at the same time. Also, it can be much faster than a naive algorithm if the number of bits in input numbers are small. We have only shown efficiency of simple pairwise addition by the BPBC technique, but we expect that other arithmetic operations can be done faster than conventional methods. We are planning to use the BPBC technique to implement small processing

TABLE III
THE RUNNING TIME (PER STRING IN MICROSECONDS) OF THE BITWISE-PARALLEL CKY PARSING FOR STRINGS OF LENGTH 32

	32 non-terminals 2097152 strings			64 non-terminals 1048576 strings			128 non-terminals 524288 strings			256 non-terminals 262144 strings			512 non-terminals 131072 strings		
p_2	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up	CPU	GPU	spd-up
32	4.62	1.24	3.73	-	-	-	-	-	-	-	-	-	-	-	-
64	7.75	1.22	6.38	8.24	2.25	3.66	-	-	-	-	-	-	-	-	-
128	12.4	1.13	11.0	13.5	2.16	6.25	16.6	3.34	4.98	-	-	-	-	-	-
256	23.1	0.956	24.2	23.8	1.58	15.0	27.1	2.77	9.78	33.5	5.45	6.15	-	-	-
512	41.0	0.827	49.5	42.0	1.54	27.3	48.1	3.48	13.8	55.2	7.64	7.23	65.6	12.8	5.11
1024	78.1	0.767	102	76.1	1.48	51.6	83.6	3.99	21.0	95.0	9.80	9.70	107	16.1	6.63
2048	156	1.35	115	151	1.80	83.7	156	4.65	33.6	169	13.1	12.8	184	21.2	8.67
4096	310	5.56	55.7	296	3.36	88.3	295	5.02	58.7	306	18.9	16.2	329	32.7	10.1
8192	626	1.84	341	588	3.68	160	574	6.71	85.5	588	33.8	17.4	605	54.2	11.2
16384	1310	3.01	436	1740	6.06	288	1910	10.7	178	2010	62.4	32.2	2070	97.6	21.2
32768	2730	6.00	454	3510	10.8	325	3850	18.6	207	4070	125	32.6	4180	188	22.2
65536	-	-	-	7010	20.4	344	7690	34.1	226	8140	248	32.8	8320	370	22.5
131072	-	-	-	14100	39.5	357	16200	65.1	249	16900	501	33.8	16900	736	23.0

cores, say, 10-bit processors. By the BPBC technique, only one thread can simulate thirty two 10-bit processor cores.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 1–15.

[6] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2012, pp. 94–102.

[7] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

[8] 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.

[9] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[10] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[12] D. Takafuji, K. Nakano, and Y. Ito, "A CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[13] T. Fujita, D. Nishikori, K. Nakano, and Y. Ito, "Efficient gpu implementations for the conway's game of life," in *Proc. International Symposium on Computing and Networking*, Dec. 2015, pp. 11–20.

[14] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, Oct. 1970.

[15] J. C. Martin, *Introduction to languages and the theory of computation (2nd Edition)*. Mac-Graw Hill, 1996.

[16] A. V. Aho and J. D. Ullman, *The Theory of Parsing Translation and Compiling*. Prentice Hall, 1972.

[17] J. L. Bordim, Y. Ito, and K. Nakano, "Accelarating the CKY parsing using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, 2003.

[18] 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, Apr. 2014.

[19] E. Charniak, *Statistical Language Learning*. Cambridge, Massachusetts: MIT Press, 1993. [Online]. Available: citeseer.nj.nec.com/charniak93statistical.html

[20] M. P. van Lohuizen, "Survey on parallel context-free parsing techniques," Delft University of Technology, Tech. Rep. IMPACT-NLI-1997-1, 1997.

[21] 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, pp. 5112–5120, 1994.

[22] J. Chang, O. Ibarra, and M. Palis, "Parallel parsing on a one-way array of finite-state machines," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 64–75, 1987.

[23] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[24] S. R. Kosaraju, "Speed of recognition of context-free languages by array automata," *SIAM J. on Computers*, vol. 4, pp. 331–340, 1975.

[25] 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.

[26] 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.

[27] ——, "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)*, 2001, pp. 590–594.

[28] Y. Yi, C.-Y. Lai, S. Petrov, and K. Keutzer, "Efficient parallel CKY parsing on GPUs," in *Proc. of International Conference on Parsing Technologies*, 2011, pp. 175–185.

[29] K.-H. Kim, S.-M. Choi, H. Lee, K. L. Man, and Y.-S. Han, "Parallel CYK membership test on GPUs," in *Proc. of International Conference on Network and Parallel Computing (LNCS 8707)*, Sept. 2014, pp. 157–168.

[30] K. Nakano, "Simple memory machine models for GPUs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

[31] ——, "Sequential memory access on the unified memory machine with application to the dynamic programming," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 85–94.