Modification of Valiant's Parsing Algorithm for the String-Searching Problem*

Yuliya Susanina[1,2][0000-0003-3904-3764], Anna Yaveyn[1][0000-0001-9733-5429], and Semyon Grigorev[1,2][0000-0002-7966-0698]

[1] Saint Petersburg State University, Universitetskaya nab. 7/9
St. Petersburg, 199034 Russia
[2] JetBrains Research, Primorskiy prospekt 68-70, Building 1
St. Petersburg 197374, Russia
jsusanina@gmail.com, anya.yaveyn@yandex.ru, semen.grigorev@jetbrains.com

Abstract. Some string-matching problems can be reduced to parsing: verification whether some sequence can be derived in the given grammar. To apply parser-based solutions to such area as bioinformatics, one needs to improve parsing techniques so that the processing of large amounts of data was possible. The most asymptotically efficient parsing algorithm that can be applied to any context-free grammar is a matrix-based algorithm proposed by Valiant. This paper presents a modification of the Valiants algorithm, which facilitates efficient utilization of modern hardware in highly-parallel implementation. Moreover, the modified version significantly decreases the number of excessive computations, accelerating the search of substrings.

Keywords: Context-free grammar · Parsing · Valiant's algorithm · String-matching · Secondary structure.

1 Introduction

The secondary structure of RNA's is tightly related to the biological functions of organisms and plays an important role in classification and recognition problems. One of the approaches to analyze RNA's secondary structure is based on formal language methods. Namely, one can process RNA sequence as a string over 4-letter alphabet $\{G, A, C, U\}$ and use formal language methods to describe properties of this string and parsing methods to analyze strings w.r.t described properties.

One of the most popular languages in this area is a context-free languags (CFL), and related probabilistic context-free grammars which widely used for secondary structure description and related tasks [8, 3]. But more powerful languages are required to describe some important features of secondary structure.

* The research was supported by the Russian Science Foundation, grant No. 18-11-00100.

For example, pseudoknots can not be expressed in terms of context-free grammar, but can be expressed by using conjunctive grammars [14] which proposed by Alexander Okhotin [10] and are the natural extensions of CFG.

For some problems, it is necessary to find all derivable substrings of the given string [4]. This case is the string-matching problem also known as a string-searching problem. The classical example of such a problem is to find substrings satisfied with the given regular expression (or regular template). But if one tries to find substring with a specific secondary structure, then it is necessary to use at least context-free template (context-free grammar) and, as a result, utilize respective parsing algorithm.

Most CFG-based approaches suffer the same issue: the computational complexity is poor. Traditionally used CYK [7, 13] runs with a cubic time complexity and demonstrates poor performance on long strings or big grammars [9]. We argue that more efficient algorithms are needed in such a field as bioinformatics where a large amount of data is common.

Asymptotically most efficient parsing algorithm is Valiant's algorithm [12] which is based on matrix multiplication. Okhotin generalized this algorithm to conjunctive and Boolean grammars [11]. Moreover, in comparison to CYK, Valiants algorithm simplifies the utilization of parallel techniques to improve performance by offloading critical computations onto matrices multiplication. However, this algorithm is not suitable for the string-matching problem.

In this paper we present the modification of Valiant's algorithm, which improves the utilization of GPGPU and parallel computations by processing some submatrices products concurrently. Also, the proposed algorithm can be easily utilized for the string-matching problem. We also prove the correctness of our algorithm and analyze its time complexity. The performance of the proposed solution was evaluated using fast matrix multiplication algorithms and parallel techniques.

2 Background

We start by introducing some basic definitions from the formal language theory. Using these definitions we describe Valiant's parsing algorithm on which we base our modification. Valiant's algorithm offloads the most time-consuming operations to operations over matrices, which makes it the most asymptotically efficient parsing algorithm with the possibility of high-performance implementation.

2.1 Formal languages

An alphabet Σ is a finite nonempty set of symbols. Σ^* is a set of all finite strings over Σ. A context-free grammar G_S is a quadruple (Σ, N, R, S), where Σ is a finite set of terminals, N is a finite set of nonterminals, R is a finite set of productions of the form $A \to \beta$, where $\Sigma \cap N = \varnothing$, $A \in N, \beta \in V^*$, $V = \Sigma \cup N$ and $S \in N$ is a start symbol. Context-free grammar $G_S = (\Sigma, N, R, S)$ is said to be in Chomsky normal form if all productions in R are of the form: $A \to BC$, $A \to a$, or $S \to \varepsilon$, where $A, B, C \in N, a \in \Sigma, \varepsilon$ is an empty string. For each context-free grammar G of length N we can find an equivalent grammar in Chomsky normal form with length N^2 [6].

$L_G(S) = \{\omega \mid S \xrightarrow[G_S]{*} \omega\}$ is a language specified by the grammar $G_S = (\Sigma, N, R, S)$, where $S \xrightarrow[G_S]{*} \omega$ means that ω can be derived in a finite number of rules applications from the start symbol S.

2.2 Valiant's parsing algorithm

Tabular parsing algorithms construct a matrix T, cells of which are filled with nonterminals from which the corresponding substring can be derived. These algorithms usually work with the grammar in Chomsky normal form. For $G_S = (\Sigma, N, R, S)$, $T_{i,j} = \{A \mid A \in N, a_{i+1} \dots a_j \in L_G(A)\} \quad \forall i < j$.

The parsing matrix T are filled successively starting with diagonal element $T_{i-1,i} = \{A \mid A \to a_i \in R\}$. Then, $T_{i,j} = f(P_{i,j})$, where $P_{i,j} = \bigcup\limits_{i<k<j} T_{i,k} \times T_{k,j}$ (here, $\times$ is the Cartesian product of two sets) and $f(P) = \{A \mid \exists A \to BC \in R : (B, C) \in P\}$. Finally, the input string $a_1 a_2 \dots a_n$ belongs to $L_G(S)$ iff $S \in T_{0,n}$.

If all cells are filled sequentially, the time complexity of this algorithm is $O(n^3)$. Valiant proposed to offload the most intensive computations to the Boolean matrix multiplication. The most time-consuming is computing $\bigcup\limits_{i<k<j} T_{i,k} \times T_{k,j}$ and Valiant's idea is to compute $T_{i,j}$ by multiplication of submatrices of T.

Multiplication of two submatrices of parsing table T is defined as follows. Let $X \in (2^N)^{m \times l}$ and $Y \in (2^N)^{l \times n}$ be two submatrices of the parsing table T. Then, denote $X \times Y = Z$, where $Z \in (2^{N \times N})^{m \times n}$ and $Z_{i,j} = \bigcup\limits_{1 \le k \le l} X_{i,k} \times Y_{k,j}$.

Note that the computation of $X \times Y$ can be replaced by the $|N|^2$ multiplication of $|N|$ Boolean matrices (for each nonterminal pair). Denote the matrix corresponding to the pair $(B, C) \in N \times N$ as $Z^{(B,C)}$, then $Z^{(B,C)}_{i,j} = 1$ iff $(B, C) \in Z_{i,j}$. It should also be noted that $Z^{(B,C)} = X^B \times Y^C$. Each Boolean matrix multiplication can be computed independently. Following these changes, time complexity of this algorithm is $O(|G|\text{BMM}(n) log(n))$ for an input string of length n, where $\text{BMM}(n)$ is the number of operations needed to multiply two Boolean matrices of size $n \times n$.

Valiant's algorithm written as proposed by Okhotin is presented in Listing 1. All elements of T and P are initialized by empty sets. Then, the elements of these two table are successively filled by two recursive procedures.

The procedure *compute*(l, m) computes values of $T_{i,j}$ for all $l \le i < j < m$.

The procedure *complete*(l, m, l', m') constructs the submatrix $T_{i,j}$ for all $l \le i < m$, $l' \le j < m'$. This procedure assumes $T_{i,j}$ for all $l \le i < j < m, l' \le i < j < m'$ are already constructed and the current value of $P_{i,j} = \{(B, C) \mid \exists k, (m \le k < l'), a_{i+1} \dots a_k \in L(B), a_{k+1} \dots a_j \in L(C)\}$ for all $l \le i < m, l' \le j < m'$. The submatrix partition during the procedure call is shown in Figure 1.

Listing 1: Parsing by Matrix Multiplication: Valiant's Version

Input: Grammar $G = (\Sigma, N, R, S), w = a_1 \dots a_n, n \geq 1, a_i \in \Sigma$, where $n + 1 = 2^p$

main():
compute(0, n + 1);
accept if and only if $S \in T_{0,n}$

compute(l, m):
if $m - l \geq 4$ **then**
    compute(l, $\frac{l+m}{2}$*);*
    compute($\frac{l+m}{2}$*, m)*
complete(l, $\frac{l+m}{2}$, $\frac{l+m}{2}$*, m)*

complete(l, m, l', m'):
if $m - l = 4$ *and* $m = l'$ **then** $T_{l,l+1} = \{A \mid A \to a_{l+1} \in R\}$;
else if $m - l = 1$ *and* $m < l'$ **then** $T_{l,l'} = f(P_{l,l'})$;
else if $m - l > 1$ **then**
    $leftgrounded = (l, \frac{l+m}{2}, \frac{l+m}{2}, m), rightgrounded = (l', \frac{l'+m'}{2}, \frac{l'+m'}{2}, m')$,
    $bottom = (\frac{l+m}{2}, m, l', \frac{l'+m'}{2}), left = (l, \frac{l+m}{2}, l', \frac{l'+m'}{2})$,
    $right = (\frac{l+m}{2}, m, \frac{l'+m'}{2}, m'), top = (l, \frac{l+m}{2}, \frac{l'+m'}{2}, m')$;
    complete(bottom);
    $P_{left} = P_{left} \cup (T_{leftgrounded} \times T_{bottom})$;
    complete(left);
    $P_{right} = P_{right} \cup (T_{bottom} \times T_{rightgrounded})$;
    complete(right);
    $P_{top} = P_{top} \cup (T_{leftgrounded} \times T_{right})$;
    $P_{top} = P_{top} \cup (T_{left} \times T_{rightgrounded})$;
    complete(top)

A simple example of the beginning of Valiant's algorithm is presented in Figure 3. Only several steps are shown, but it is enough to compare our version with the original algorithm.

3 Modified Valiant's algorithm

In this section, we propose how to rearrange the order in which submatrices are processed in the algorithm. The different order improves the independence of submatrices handling and facilitates the implementation of parallel submatrix processing.

3.1 Layered submatrices processing

We propose to divide the parsing table into layers of disjoint submatrices of the same size (see Figure 2). Such division is possible because the derivation of a substring of the fixed length does not depend on either left or right contexts.

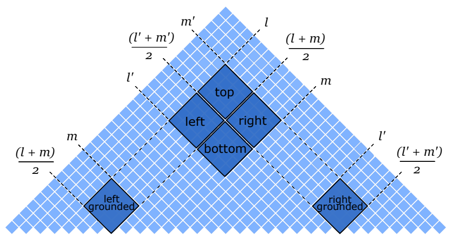

Fig. 1. Matrix partition used in procedure *complete(l, m, l', m')*

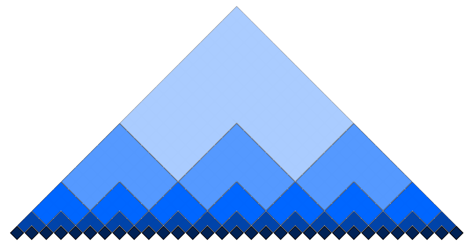

Fig. 2. Matrix partition on V-shaped layers used in modification

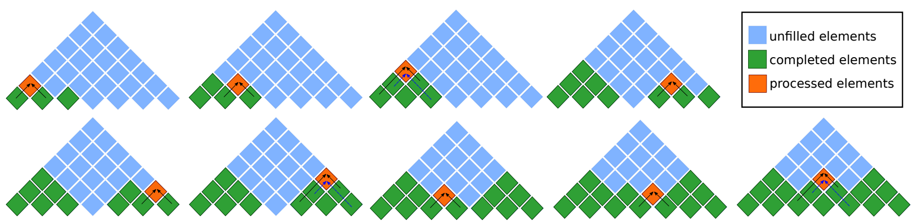

Fig. 3. An example of the beginning of Valiant's algorithm

Each layer consists of square matrices which size is a power of 2. The layers are computed successively in the bottom-up order. Each matrix in the layer can be handled independently, which facilitates parallelization of layer processing.

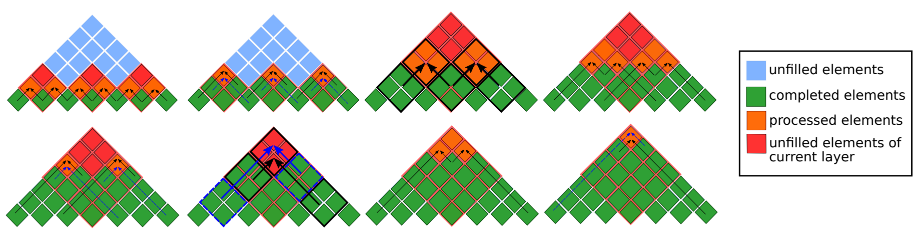

Fig. 4. An example of the modification of Valiant's algorithm

Figure 4 demonstrates the modified algorithm. The lowest layer (submatrices of size 1) has already been computed. The second layer is filled in the steps 1-2. So the same part of parsing matrix (as in Figure 3) can be computed only in two steps using parallel computation of submatrix products.

The modified version of Valiant's algorithm is presented in Listing 2. The procedure *main()* computes the lowest layer ($T_{l,l+1}$), and then divides the table into layers, and computes them with the *completeVLayer()* function. Thus, *main()* computes all elements of parsing table T.

We define *left(subm)*, *right(subm)*, *top(subm)*, *bottom(subm)*, *rightgrounded(subm)* and *leftgrounded(subm)* functions which return the submatrices for matrix $subm = (l, m, l', m')$ according to the original Valiant's algorithm (Figure 2).

Listing 2: Parsing by Matrix Multiplication: Modified Version

Input: $G = (\Sigma, N, R, S), w = a_1 \dots a_n, n \geq 1, n+1 = 2^p, a_i \in \Sigma$

main():

for $l \in \{1, \dots, n\}$ **do** $T_{l,l+1} = \{A | A \to a_{l+1} \in R\}$;
for $1 \leq i \leq p-1$ **do**
  layer = constructLayer(i);
  completeVLayer(layer)
accept if and only if $S \in T_{0,n}$

constructLayer(i):

$\{(k2^i, (k+1)2^i, (k+1)2^i, (k+2)2^i) \mid 0 \leq k < 2^{p-i} - 1\}$

completeLayer(M):

if $\forall (l, m, l', m') \in M \quad (m - l = 1)$ **then**
  for $(l, m, l', m') \in M$ **do** $T_{l,l'} = f(P_{l,l'})$;
else
  completeLayer($\{$*bottom(subm)* $\mid$ *subm* $\in M\}$);
  completeVLayer(M)

completeVLayer(M):

multiplicationTasks$_1$ =
 $\{$*left(subm), leftgrounded(subm), bottom(subm)* $\mid$ *subm* $\in M\} \cup$
 $\{$*right(subm), bottom(subm), rightgrounded(subm)* $\mid$ *subm* $\in M\}$;
multiplicationTask$_2$ = $\{$*top(subm), leftgrounded(subm), right(subm)* $\mid$ *subm* $\in M\}$;
multiplicationTask$_3$ = $\{$*top(subm), left(subm), rightgrounded(subm)* $\mid$ *subm* $\in M\}$;
performMultiplications(multiplicationTask$_1$*)*;
completeLayer($\{$*left(subm)* $\mid$ *subm* $\in M\} \cup \{$*right(subm)* $\mid$ *subm* $\in M\}$);
performMultiplications(multiplicationTask$_2$*)*;
performMultiplications(multiplicationTask$_3$*)*;
completeLayer($\{$*top(subm)* $\mid$ *subm* $\in M\}$)

performMultiplication(tasks):

for $(m, m1, m2) \in$ *tasks* **do** $P_m = P_m \cup (T_{m1} \times T_{m2})$;

The procedure *completeVLayer(M)* takes an array of disjoint submatrices M which represents a layer. For each *subm = (l, m, l', m')* $\in M$ this procedure computes *left(subm), right(subm), top(subm)*. The procedure assumes that the elements of *bottom(subm)* and $T_{i,j}$ for all i and j such that $l \leq i < j < m$ and $l' \leq i < j < m'$ are already constructed. Also it is assumed that the current value of $P_{i,j} = \{(B, C) \mid \exists k, (m \leq k < l'), a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ for all i and j such that $l \leq i < m$ and $l' \leq j < m'$.

The procedure *completeLayer(M)* takes an array of disjoint submatrices M, but unlike the previous one, it computes $T_{i,j}$ for all $(i, j) \in subm$. This procedure requires the same assumptions on $T_{i,j}$ and $P_{i,j}$ as in the original algorithm.

In other words, *completeVLayer(M)* computes the entire layer M and *completeLayer(M_2)* is a helper function which is necessary for computation of smaller square submatrices $subm_2 \in M_2$, where M_2 is a sublayer of M.

Finally, the procedure *performMultiplication(tasks)*, where *tasks* is an array of triples of submatrices, performs the basic step of the algorithm: matrix multiplication. It is worth mentioning that $|tasks| \geq 1$ and each task can be computed independently, while the original algorithm handles one *task* per step sequentially. So, the practical implementation of this procedure can easily utilize different techniques of parallel array processing.

3.2 Correctness and complexity

We provide the proof of correctness and time complexity of the proposed modification in this section.

Lemma 1. *Let M be a layer. If for all $(l, m, l', m') \in M$:*

1. $T_{i,j} = \{A \mid a_{i+1} \dots a_j \in L_G(A)\}$ *for all i and j such that $l \leq i < j < m$ and $l' \leq i < j < m'$;*
2. $P_{i,j} = \{(B, C) \mid \exists k, (m \leq k < l') : a_{i+1} \dots a_k \in L_G(B), a_{k+1} \dots a_j \in L_G(C)\}$ *for all $l \leq i < m$ and $l' \leq j < m'$.*

Then the procedure completeLayer(M), returns correctly computed sets of $T_{i,j}$ for all $l \leq i < m$ and $l' \leq j < m'$ for all $(l, m, l', m') \in M$.

Proof. The algorithm from listing 2 is given by mutually recursive functions *completeLayer()* and *completeVLayer()*. Next we show, that if all necessary conditions are met, *completeVLayer(M)* return correctly computed layer M.

We consider the matrix of size $m - l = 1$ (and then the elements of layer M are correctly computed in lines 10-11) and thereafter the lemma can be proved by induction by $m - l$.

Theorem 1. *Algorithm from listing 2 correctly computes $T_{i,j}$ for all i and j, thus an input string $a = a_1 a_2 \dots a_n \in L_G(S)$ if and only if $S \in T_{0,n}$.*

Proof. Primarily to prove the theorem, we show by induction that all layers of the parsing table T are computed correctly.

Basis: layer of size 1×1. Parsing table T consists of one layer of size 1 and its elements are correctly computed in lines 2-3 in listing 2.

Inductive step: assume any layer of size less than or equal to $2^{r-2} \times 2^{r-2}$ are computed correctly.

Define layer of size $2^{r-1} \times 2^{r-1}$ as M. Hereinafter *subm = (l, m, l', m')* is a typical element of layer M.

Consider *completeVLayer(M)* call.

Firstly, *performMultiplications(multiplicationTask$_1$)* adds to each $\mathrm{P}_{i,j}$ all pairs (B,C) such that $\exists k$, $(\frac{l+m}{2} \leq k < l')$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ for all $(i,j) \in leftsublayer(M)$ and (B,C) such that $\exists k$, $(m \leq k < \frac{l'+m'}{2})$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ for all $(i,j) \in rightsublayer(M)$. Now *completeLayer(leftsublayer(M) ∪ rightsublayer(M))* can be called and it returns correctly computed *leftsublayer(M) ∪ rightsublayer(M)*.

Then *performMultiplications* called with arguments *multiplicationTask$_2$* and *multiplicationTask$_3$* adds pairs (B,C) such that $\exists k$, $(\frac{l+m}{2} \leq k < m)$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ and pairs (B,C) such that $\exists k$, $(l' \leq k < \frac{l'+m'}{2})$, $a_{i+1} \dots a_k \in L_G(B)$, $a_{k+1} \dots a_j \in L_G(C)$ to each element $\mathrm{P}_{i,j}$ for all $(i,j) \in topsublayer(M)$. So as $m = l'$ (from the construction of the layer), condition for elements of matrix P are fulfilled. Now *completeLayer(topsublayer(M))* can be called and it returns correctly computed *topsublayer(M)*.

All $T[i,j]$ $\forall (i,j) \in M$ are computed correctly.

Thus, *completeVLayer(M)* returns correct $T_{i,j}$ for all $(i,j) \in M$ for any layer M of parsing table T and lines 4-6 in listing 2 return all $T_{i,j} = \{A \mid A \in N, a_{i+1} \dots a_j \in L_G(A)\}$.

Lemma 2. *Let calls$_r$ is a number of the calls of completeVLayer(M) where for all $(l,m,l',m') \in M$ with $m - l = 2^{p-r}$.*

- *for all $r \in \{1,..,p-1\}$ $\sum_{n=1}^{calls_r} |M|$ is exactly $2^{2r-1} - 2^{r-1}$;*
- *for all $r \in \{1,..,p-1\}$ products of submatrices of size $2^{p-r} \times 2^{p-r}$ are calculated exactly $2^{2r-1} - 2^r$ times.*

Proof. Prove the first statement by induction on r.

Basis: $r = 1$. $calls_1$ and $|M| = 1$. So, $2^{2r-1} - 2^{r-1} = 2^1 - 2^0 = 1$.

Inductive step: assume that $\sum_{n=1}^{calls_r} |M|$ is exactly $2^{2r-1} - 2^{r-1}$ for all $r \in \{1,..,q\}$.

Let us consider $r = q + 1$.

Firstly, note that function *costructLayer(r)* returns $2^{p-r} - 1$ matrices of size 2^r, so in the call of *completeVLayer(costructLayer(p - r)) costructLayer(p - r)* returns $2^r - 1$ matrices of size 2^{p-r}. Secondly, *completeVLayer()* is called 3 times for the left, right and top submatrices of size $2^{p-(r-1)}$. Finally, *completeVLayer()* is called 4 times for the bottom, left, right and top submatrices of size $2^{p-(r-2)}$, except $2^{r-2} - 1$ matrices which were already computed.

Then, $\sum_{n=1}^{calls_r} |M| = 2^r - 1 + 3 \times (2^{2(r-1)-1} - 2^{(r-1)-1}) + 4 \times (2^{2(r-2)-1} - 2^{(r-2)-1}) - (2^{r-2} - 1) = 2^{2r-1} - 2^{r-1}$.

Now we know $\sum_{n=1}^{calls_{r-1}} |M|$ is $2^{2(r-1)-1} - 2^{(r-1)-1}$ and we can calculate the number of products of submatrices of size $2^{p-r} \times 2^{p-r}$. During these calls *performMultiplications* runs 3 times, $|multiplicationTask1| = 2 \times 2^{2(r-1)-1} - 2^{(r-1)-1}$ and $|multiplicationTask2| = |multiplicationTask3| = 2^{2(r-1)-1} - 2^{(r-1)-1}$. So, the number of products of submatrices of size $2^{p-r} \times 2^{p-r}$ is $4 \times (2^{2(r-1)-1} - 2^{(r-1)-1}) = 2^{2r-1} - 2^r$.

Theorem 2. *Let $|G|$ be a length of the description of the grammar G and let n be a length of an input string. Then algorithm from listing 2 calculates matrix T*

in $\mathcal{O}(|G|BMM(n)\log n)$ where BMM(n) is the number of operations needed to multiply two Boolean matrices of size $n \times n$.

Proof. The proof is almost identical with the proof of theorem 1 given by Okhotin [11], because, as shown in the last lemma, the Algorithm 1 has the same number of products of submatrices.

To summarize, the correctness of the modification was proved and it was shown that the time complexity remained the same as in Valiant's version.

3.3 Algorithm for substrings

Next, we show how our modification can be applied to the string-matching problem. To find all substrings of size s, which can be derived from a start symbol for an input string of size $n = 2^p - 1$, we need to compute layers with submatrices of size not greater than 2^r, where $2^{r-2} < s \leq 2^{r-1}$.

Let $r = p - (m-2)$ and consequently $(m-2) = p - r$. For any $m \leq i \leq p$ products of submatrices of size 2^{p-i} are calculated exactly $2^{2i-1} - 2^i$ times and each of them imply multiplying $C = \mathcal{O}(|G|)$ Boolean submatrices. Let $\mathrm{BMM} = n^\omega f(n)$, where $\omega \geq 2$ and $f(n) = n^{o(1)}$. Now we estimate the number of operations needed to find all substrings:

$$C \cdot \sum_{i=m}^{p} 2^{2i-1} \cdot 2^{\omega(p-i)} \cdot f(2^{p-i}) = C \cdot 2^{\omega r} \sum_{i=2}^{r} 2^{(2-\omega)i} \cdot 2^{2(p-r)-1} \cdot f(2^{r-i}) \leq$$
$$C \cdot 2^{\omega r} f(2^r) \cdot 2^{2(p-r)-1} \sum_{i=2}^{r} 2^{(2-\omega)i} = \mathrm{BMM}(2^r) \cdot 2^{2(p-r)-1} \sum_{i=2}^{r} 2^{(2-\omega)i}$$

Thus, time complexity for searching all substrings of size not greater than 2^r is $O(2^{2(p-r)-1}|G|\mathrm{BMM}(2^r)(r-1))$ where the appeared factor meet the number of matrices in the last completed layer, while time complexity for the full input string is $O(|G|\mathrm{BMM}(2^p)(p-1))$. The Valiant's algorithm completely calculate at least 2 triangle submatrices of size $\frac{n}{2}$, as shown in Figure 5, thus the minimum asymptotic complexity is $O(|G|\mathrm{BMM}(2^{p-1})(p-2))$. Thus we can conclude that the modification is asymptotically faster than the original algorithm for substrings of size $s \ll n$.

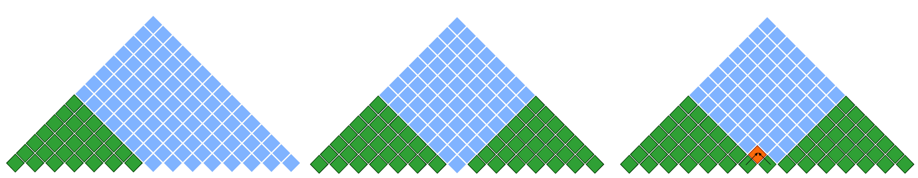

Fig. 5. The number of elements necessary to compute in Valiant's algorithm. It is necessary to calculate at least 2 triangle submatrices of size $\frac{n}{2}$.

4 Evaluation

In this section, we present the results of experiments whose purpose is to demonstrate the practical applicability of the proposed algorithm.

All tests were run on a PC with the following characteristics: OS: Linux Mint 19.1, CPU: Intel i5-8250U, 1600-3400 Mhz, RAM: 8 GB, GPU: NVIDIA GeForce GTX 1050 MAX-Q.

We implement two different versions of Valiant's algorithm and its modification in C++ programming language (the source code is available on GitHub[3]):

- CPU-based solutions (valCPU and modCPU):
  A high-performance library M4RI [1] for fast Boolean matrix multiplication is used. This library include one of the most efficient implementations of the Method of the Four Russians [2].
- GPU-based solutions (valGPU and modGPU):
  A naive Boolean matrix multiplication in CUDA C with Boolean values treated as bits and packed into `uint_32` is implemented.

We evaluate these implementations on context-free grammars D_2 (Figure 6) and BIO (Figure 7). Grammar D_2 generates Dyck language on two types of parentheses and is chosen because grammars that describe well-balanced sequences of brackets are often used in string analysis in bioinformatics. Grammar BIO applies to the tRNA classification problem in paper [5]. We test various strings with length n up to 8191 and search substrings with length $subs$ up to 2040.

$$s \to s\ s \mid (\ s\ ) \mid [\ s\ ] \mid \varepsilon$$

Fig. 6. Grammar D_2.

$$\begin{aligned}
s &\to \text{stem}\langle s0\rangle \\
\text{any_str} &\to \text{any_smb} * [2..10] \\
s0 &\to \text{any_str} \mid \text{any_str stem}\langle s0\rangle\ s0 \\
\text{any_smb} &\to A \mid U \mid C \mid G \\
\text{stem1}\langle s1\rangle &\to A\ s1\ U \mid G\ s1\ C \mid U\ s1\ A \mid C\ s1\ G \\
\text{stem2}\langle s1\rangle &\to \text{stem1}\langle \text{stem1}\langle s1\rangle\rangle \\
\text{stem}\langle s1\rangle &\to A\ \text{stem}\langle s1\rangle\ U \mid U\ \text{stem}\langle s1\rangle\ A \\
&\quad \mid C\ \text{stem}\langle s1\rangle\ G \mid G\ \text{stem}\langle s1\rangle\ C \\
&\quad \mid \text{stem1}\langle \text{stem2}\langle s1\rangle\rangle
\end{aligned}$$

Fig. 7. Grammar BIO.

The results of the evaluation are summarized in the tables below. Time is measured in milliseconds.

[3] GitHub repository of the YaccConstructor project: https://github.com/YaccConstructor/YaccConstructor.

Table 1. Results of the comparison of Valiant's algorithm and the modification

n	Grammar D_2				Grammar BIO			
	valCPU	modCPU	valGPU	modGPU	valCPU	modCPU	valGPU	modGPU
127	78	76	195	105	1345	1339	193	106
255	289	292	523	130	5408	5488	525	140
511	1212	1177	1909	250	21969	22347	1994	256
1023	4858	4779	7878	540	88698	90318	7890	598
2047	19613	19379	33508	1500	363324	374204	34010	1701
4095	78361	78279	140473	4453	1467675	1480594	141104	5472
8191	315677	315088	-	13650	-	-	-	18039

The comparative analysis (table 1) shows that the performance of Valiant's and modified algorithms is the same for the CPU-based solutions. GPU-based implementation of Valiant algorithm is slower for grammar D_2 than the CPU-based one. It is probably because of processing a large amount of small matrix multiplication which cannot be computed concurrently. In other cases, GPU-based solution provides significant performance improvement, especially for our modification, where the utilization of parallelism works for matrix multiplication itself as well as each matrix in the layer.

To adapt our algorithm for the string-matching problem the $main()$ function takes an additional argument sub — the maximum length of strings we want to find, so the modification has no need to compute all layers as shown in section 3.3. The corresponding implementations denote as `adpCPU` and `adpGPU`.

Table 2. Results of applying the modification to the string-searching problem

sub	n	Grammar BIO	
		adpCPU	adpGPU
250	1023	2996	242
	2047	6647	255
	4095	13825	320
	8191	28904	456
510	2047	12178	583
	4095	26576	653
	8191	56703	884
1020	4095	48314	1590
	8191	108382	1953
2040	4095	197324	5100

The results of the second evaluation (table 2) shows that the modified version of algorithm can find all derivable substrings much faster than Valiant's algorithm, so can be efficiently applied to the string-searching problem.

5 Conclusion and future works

We presented a modification of Valiant's algorithm which makes it possible to process each matrix in layer independently and use parallel computations more efficiently. This new algorithm can efficiently handle the problem of finding all substrings of a specified length. The proposed algorithm is accompanied by proof of correctness and computational complexity analysis. We showed the practical applicability of our modification. Concurrent processing of matrices in layer significantly increases the performance of GPU-based solution. Also, the modification can find all substrings much faster than Valiant's algorithm through the possibility to stop parsing matrix filling.

The directions for future research is to extend the proposed algorithm for conjunctive and Boolean grammars handling. It will be useful for complex secondary structure features processing.

References

1. Albrecht, M., Bard, G., Hart, W.: Algorithm 898. ACM Transactions on Mathematical Software **37**(1), 114 (Jan 2010). https://doi.org/10.1145/1644001.1644010, http://dx.doi.org/10.1145/1644001.1644010
2. Arlazarov, V.L., Dinitz, Y.A., Kronrod, M., Faradzhev, I.: On economical construction of the transitive closure of an oriented graph. In: Doklady Akademii Nauk. vol. 194, pp. 487–488. Russian Academy of Sciences (1970)
3. Dowell, R.D., Eddy, S.R.: Evaluation of several lightweight stochastic context-free grammars for rna secondary structure prediction. BMC bioinformatics **5**(1), 71 (2004)
4. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological sequence analysis. Cambridge University Press (1996)
5. Grigorev., S., Lunina., P.: The composition of dense neural networks and formal grammars for secondary structure analysis. In: Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3 BIOINFORMATICS: BIOINFORMATICS,. pp. 234–241. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007472302340241
6. Hopcroft, J.E.: Introduction to automata theory, languages, and computation. Pearson Education India (2008)
7. Kasami, T.: An efficient recognition and syntax-analysis algorithm for context-free languages. Coordinated Science Laboratory Report no. R-257 (1966)
8. Knudsen, B., Hein, J.: Rna secondary structure prediction using stochastic context-free grammars and evolutionary history. Bioinformatics (Oxford, England) **15**(6), 446–454 (1999)
9. Liu, T., Schmidt, B.: Parallel rna secondary structure prediction using stochastic context-free grammars. Concurrency and Computation: Practice and Experience **17**(14), 1669–1685 (2005)
10. Okhotin, A.: Conjunctive grammars. J. Autom. Lang. Comb. **6**(4), 519–535 (Apr 2001)
11. Okhotin, A.: Parsing by matrix multiplication generalized to boolean grammars. Theor. Comput. Sci. **516**, 101–120 (Jan 2014), http://dx.doi.org/10.1016/j.tcs.2013.09.011

12. Valiant, L.G.: General context-free recognition in less than cubic time. J. Comput. Syst. Sci. **10**(2), 308–315 (Apr 1975), http://dx.doi.org/10.1016/S0022-0000(75)80046-8
13. Younger, D.H.: Context-free language processing in time n3. In: Proceedings of the 7th Annual Symposium on Switching and Automata Theory (Swat 1966). pp. 7–20. SWAT '66, IEEE Computer Society, Washington, DC, USA (1966), https://doi.org/10.1109/SWAT.1966.7
14. Zier-Vogel, R., Domaratzki, M.: Rna pseudoknot prediction through stochastic conjunctive grammars. Computability in Europe 2013. Informal Proceedings pp. 80–89 (2013)