Parallel CYK Membership Test on GPUs

Kyoung-Hwan Kim[1], Sang-Min Choi[1], Hyein Lee[1], Ka Lok Man[2], and Yo-Sub Han[1,⋆]

[1] Department of Computer Science, Yonsei University, Seoul, Republic of Korea
{kyounghwan,jerassi,hyein,emmous}@cs.yonsei.ac.kr
[2] Department of Computer Science and Software Engineering, Xian Jiaotong-Liverpool University, Suzhou, China
ka.man@xjtlu.edu.cn

Abstract. Nowadays general-purpose computing on graphics processing units (GPGPUs) performs computations what were formerly handled by the CPU using hundreds of cores on GPUs. It often improves the performance of sequential computation when the running program is well-structured and formulated for massive threading. The CYK algorithm is a well-known algorithm for the context-free language membership test and has been used in many applications including grammar inferences, compilers and natural language processing. We revisit the CYK algorithm and its structural properties suitable for parallelization. Based on the discovered properties, we then parallelize the algorithm using different combinations of memory types and data allocation schemes using a GPU. We evaluate the algorithm based on real-world data and herein demonstrate the performance improvement compared with CPU-based computations.

Keywords: Parallel Computing, Context-Free Language Membership Test, CYK Algorithm, GPU Programming, CUDA.

1 Introduction

Graphics Processing Unit (GPU) computing involves the use of a GPU to improve general-purpose scientific applications, that were formerly handled by a CPU. A GPU consists of processors with different instruction set architectures (ISAs). GPUs designed with massively parallel single instruction multiple threads (SIMT) are many-core processors that provide an effective performance through low-latency and high-bandwidth. The limits of program scalability are often related to some combination of memory bandwidth saturation, memory contention, imbalanced data distribution or data structure/algorithm interactions. For a better performance, researchers and developers have therefore suggested particular data structures and formulated problems specifically for massive threading. They executed massive threads by leveraging shared memory resources including [7,23]. There are a few tools that support general purpose computing for GPUs such as the Compute Unified Device Architecture

⋆ Corresponding author.

C.-H. Hsu et al. (Eds.): NPC 2014, LNCS 8707, pp. 157–168, 2014.

(CUDA) [18] and Open Computing Language (OpenCL) [13]. We considered the Cocke-Younger-Kasami (CYK) algorithm [6,12,22], which is popular for several application domains such as RNA secondary structure prediction [4] and grammatical inference [17], and implement parallel CYK algorithms using GPUs. Note that the runtime of the CYK algorithm is $O(|G|n^3)$, where n is the length of an input string and $|G|$ is the size of the input context-free grammar (CFG) [1]. Namely, the runtime of the CYK algorithm is closely related to the size of the grammar and the length of the input string. We investigated the possible grammar mapping methods for hundreds of cores in a GPU, which may give rise to an overall performance improvement of the CYK algorithm. In particular, we considered three mapping methods: rule-based, left-variable sorting and right-variable sorting mappings. We applied each mapping to different architectural features of GPUs such as zero-copy host memory, page-locked memory, shared memory and texture memory. We then evaluated the algorithm for different combinations of mapping methods and features. We ran our experiments on NVIDIA GPUs (GTX560Ti) with 384 cores using the dataset from the Berkeley parser [16] and Penn Treebank [14].

In Section 2, we revisit previous research on the parallelization of the CYK algorithm. We then recall CFG and the CYK algorithm in Section 3. We describe three mapping methods and four memory structures in Section 4. We then present our experimental results and an analysis from applying the four memory structures to each mapping method in Section 5. Finally, some concluding remarks regarding this research are given in Section 6.

2 Related Work

The CYK algorithm allows us to determine whether an input string is in an input context-free language. The algorithm has been widely used in several domains such as parsing, grammatical inference and bioinformatics. The CYK algorithm is a classical dynamic programming algorithm and there have been many attempts to parallelize it depending on the applications used. Table 1 shows previous studies.

3 CFG Membership Test

We briefly recall the definition of context-free languages and the CYK algorithm. For more details on these topics, the reader is referred to Hopcroft and Ullman [10].

3.1 Context-Free Languages

A CFG G is specified by the tuple $G = (V, \Sigma, P, S)$, where V is a set of variables, Σ is a set of terminals, P is a set of production rules and S is the start symbol. Given a CFG $G = (V, \Sigma, P, S)$, let $\alpha A \beta$ be a string derived from S where $A \in V$

Table 1. Related work on the parallel CYK implementation

Authors	Year	Summary
Takashi et al. [19]	1997	They suggested an agenda-based parallel CYK parser on a distributed-memory parallel machine that consists of 256 nodes (single processors). This approach uses a specific parallel language and parallelizes the CYK algorithm by allocating each cell of the CYK matrix into a processor.
Bordim et al. [3]	2002	They studied the CYK algorithm on field programmable gate arrays (FPGAs) and developed a hardware generator that creates a Verilog HDL source performing CYK parsing for a given CFG. Their approach considers 2,048 production rules and 64 variables in an input CFG and shows a speedup factor of almost 750×.
Johnson [11]	2011	The author examined the CYK algorithm for a dense probabilistic context-free grammars (PCFG) and constructed a dense PCFG with 32 variables and 32,768 production rules with random probability. The author reported an 18.4× speedup obtained on NVIDIA Fermi s2050 GPUs, and suggested a reduction method in one block for calculating the probability.
Dunlop et al. [8]	2011	They presented a matrix encoding of CFGs using a multiplication method for a matrix low-latency parallelized CYK algorithm. They encoded the grammars of CFG in a matrix form in which the rows are the left-hand side variable of the production and the columns are the right-hand side variables (pairs in CNF).
Yi et al. [21]	2014	They proposed an efficient parallel CYK algorithm for natural language parsing of PCFG on GPUs. A PCFG is a CFG in which each production is augmented with the probability. Their algorithm assigns each production rule of an input PCFG to each core in a GPU and finds the valid parsing rules quickly.

and α and β are strings from $(V \cup \Sigma)^*$. When $A \to \gamma$, we say that A is rewritten to γ and denote this derivation step by $\Rightarrow$ symbol: namely, $\alpha A \beta \Rightarrow \alpha\gamma\beta$. When there are zero or more steps of derivation, we denote this step by $\overset{*}{\Rightarrow}$ symbol. The language $L(G)$ of G is then a set of terminal strings derived from the start symbol S; namely, $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. We can say that a CFG $G = (V, \Sigma, P, S)$ is in Chomsky Normal Form (CNF) if every production rule in P is either of form $A \to BC$ or $A \to a$, where $A, B, C \in V$ and $a \in \Sigma$ [5]. It is well-known that every CFG can be transformed in to CNF [10]. From now on, we assume that an input CFG is in CNF.

Procedure 1. CYK Algorithm

```
1: procedure CYK ALGORITHM(G = (V, Σ, P, S), w)
2:     initialize table M[n][n + 1][|V|];          ▷ |V| is size of variables
3:     n = length of input string w
4:     for i = 0 to n − 1
5:         if {A ∈ V | A → w_i ∈ P}
6:             M[i][i + 1][A] = T                  ▷ Initialize with terminal rules
7:     end for
8:     for len = 2 to n
9:         transitionRule(M, n, len, G);           ▷ Procedure 2
10:    end for
11:    if M[0][n][S] = T
12:        return true
13:    else
14:        return false
15:    end if
16: end procedure
```

3.2 CYK Algorithm

Given an input string $w = w_1 w_2 \cdots w_n \in \Sigma^*$ and a CFG $G = (V, \Sigma, P, S)$, the CYK algorithm, which is based on the bottom-up dynamic programming, determines whether w is in $L(G)$. The algorithm constructs a triangular table M in which each cell $M[i-1][j][A]$, for $A \in V$, is T if $A \stackrel{*}{\Rightarrow} w_i w_{i+1} \cdots w_j$ in G. Once all of M is computed, the algorithm checks whether $M[0][n][S] = \mathtt{T}$.

Procedure 2. transitionRule

```
1: procedure TRANSRULE(M, n, len, G = (V, Σ, P, S))
2:     for i = 0 to n − len do                     ▷ Start Index
3:         j = i + len;                            ▷ End Index
4:         foreach production A → BC ∈ P
5:             for split = i + 1 to j − 1
6:                 if M[i][split][B] =T do
7:                     if M[split][j][C] =T do
8:                         M[i][j][A] =T;
9:         end foreach
10:    end for
11: end procedure
```

First, we initialize the bottom level of the table using the terminal rules (line 5 and 6). The algorithm then proceeds to repeatedly apply all binary rules and builds up for the table using Procedure 2. As illustrated in Procedure 1, the algorithm fills up M and checks whether $M[0][n][S]$ is T.

Fig. 1 illustrates the CYK table M for the string $w = cabac$ with respect to a CFG $G = \{\{S, A, B, C\}, \{a, b, c\}, P, S\}$, where $P = \{S \to AB \mid b, A \to CB \mid AA \mid a, B \to AS \mid b, C \to BS \mid c\}$.

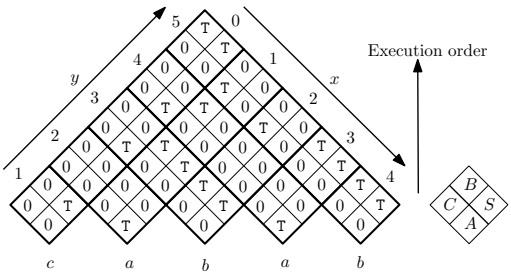

Fig. 1. An example of the CYK table for an input string, *cabab*

4 Our Approaches and Implementations

We next discuss the different implementations of Procedure 2 that account for the bulk of the overall execution time for the CYK algorithm. We consider three thread mappings, memory types for data access, and two data transfer methods.

4.1 Three Types of Thread Mappings

One of the important factors for designing parallel algorithms is how to map the input data to the threads for fast parallel processing. We consider the possible mappings of the grammar rules to the threads for the table cells or variables of the CYK algorithm. We select grammar rules for thread mapping instead of variables. Since the number of variables is usually fewer than the number of threads, it is possible to fail to provide enough parallelism to fully utilize the massive number of threads in GPUs. In addition, a load imbalance exists because of differences in the number of rules for each variables and it leads to different branches and degrades the performance. We can reduce the load imbalance by mapping the rules to the threads. There are three mapping methods used: rule-based, left-variable sorting ($\mathcal{LVS}$) and right-variable sorting ($\mathcal{RVS}$) mappings.

1. **Rule-Based Mapping:** We noticed that the **foreach** (line 4) loop in Procedure 2 is suitable for parallelization. We therefore simply map all rules in the input grammar to all available threads as described in Procedure 3.
2. $\mathcal{RVS}$ **Mapping:** The $\mathcal{RVS}$ mapping is to sort the production rules in a CFG, according to the first variable of the right-hand side (RHS) in the production rules. The left figure in Fig. 2 shows an example of $\mathcal{RVS}$. The main reason for introducing $\mathcal{RVS}$ mapping is to reduce the thread divergence. In Procedure 2, we first check whether the two variables on the RHS exist in each cell. If they exist, we store them in the current cell. Thread-divergence occurs since each rule has different first variables in the RHS. For example, when some threads that have the first RHS variable satisfying an if-condition to enter the if-statement, other threads must wait until the statement ends.

Procedure 3. RuleandRVSTR

```
1: procedure RuleandRVSTR(M, n, len, G = (V, Σ, P, S))
2:     for i = 0 to n − len do in parallel                ▷ Mapping to Thread Block
3:         j = i + len;
4:         __shared__ bool shVar[|V|];
5:         foreach production A → BC ∈ P in parallel
6:                                                       ▷ Mapping to Thread
7:             for split = i + 1 to j − 1
8:                 if M[i][split][B] = T do
9:                     if M[split][j][C] = T do
10:                        shVar[|A|]= T;
11:        end foreach
12:        for A ∈ V in parallel
13:            M[i][j][A] =shVar[|A|];
14:    end for
15: end procedure
```

This situation degrades the performance since some threads must wait for the others. Thus, in the variable-based mapping of algorithm, we avoid this type of divergence by sorting the first-right variables in the production rules.

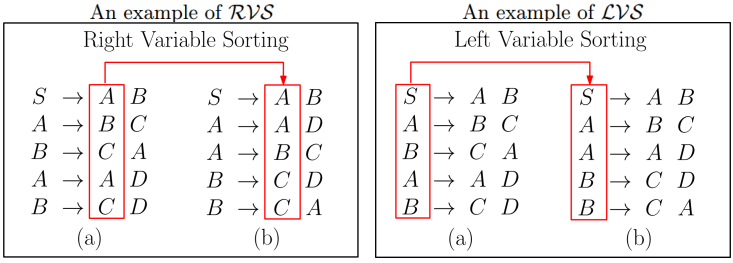

Fig. 2. An example of $\mathcal{RVS}$ and $\mathcal{LVS}$

3. $\mathcal{LVS}$ **Mapping:** The $\mathcal{LVS}$ mapping first sorts all production rules in a CFG, to the left-hand side (LHS) variable of the rules. The right figure in Fig. 2 shows examples of $\mathcal{LVS}$: (a) and (b) are the production rules before and after $\mathcal{LVS}$, respectively. We group the rules that have the same LHS variables. The purpose of $\mathcal{LVS}$ is to concurrently access variables with a set of rules that have the same LHS value. When we store the variables in Procedure 4, we aim to improve the performance by storing the sorted variables into single variable.
4. **Adding Dummy:** If we use dummy rules in $\mathcal{RVS}$ mapping, we could improve the overall performance since some threads may not need to wait on line 6 in Procedure 2. In $\mathcal{LVS}$ mapping, we use dummies because each warp has only one LHS value, which allows us to save a LHS value to one memory

Procedure 4. LVSTR

```
1: procedure LVSTR(M, n, len, G = (V, Σ, P, S))
2:     for i = 0 to n − len do in parallel                ▷ Mapping to Thread Block
3:         j = i + len;
4:         __shared__ int shVar[|W|];                     ▷ |W| is size of warps
5:         foreach production A → BC ∈ P in parallel
6:                                                        ▷ Mapping to Thread
7:             for split = i + 1 to j − 1
8:                 if M[i][split][B] = T do
9:                     if M[split][j][C] = T do
10:                        shVar[w]= T;                   ▷ w is warp number
11:        end foreach
12:        for thread t_w ∈ each warp w in parallel
13:            M[i][j][shVar[|w|]]=T;
14:    end for
15: end procedure
```

storage. By adding dummies, the memory access may be increased. However, we obtain performance improvement by reducing thread divergence and sharing one LHS value in each thread blocks. If the grammars are not ordered by LHS or RHS, the adding dummies is effective, since the performance improvement is greater than the overhead from the additional memory access. We describe this tendency and the results of adjusting this method to our implementation in the experimental results, in Section 5

4.2 Two Types of Memory for Data Access

In GPU programming, data are usually allocated and accessed in global memory. Since the access speed of global memory is slow, reducing access is important for high-performance. We therefore use texture memory and shared memory to reduce access.

- **Texture Memory:** Texture memory is read-only memory and is allocated by calling a binding API in CUDA. Unlike global memory, texture memory provides caching and reads all threads in a kernel. If the memory is frequently accessed, it becomes more efficient since it has a faster access speed than global memory. On the other hand, it has an overhead caused from binding the device data after the memory allocation is initiated from the host.
- **Shared Memory:** Shared memory is a memory block that can be accessed by all threads within a block. It is much faster than local and global memory. We use this memory in rule-based and $\mathcal{RVS}$ mapping by following two steps: First, we allocate the variables and their number to the shared memory and store them before saving them directly to the table. We then restore these variables to the global memory. These steps differ from those of Procedure 3. Since all threads in the same warp have the same LHS variable because of deploying dummy rules, we can save one variable instead of all variables in

the shared memory. Therefore, $\mathcal{LVS}$ mapping need smaller space than rule-based and $\mathcal{RVS}$ mapping. We add the following processes to Procedure 3 in order to implement $\mathcal{LVS}$ mapping; we allocate the shared memory based on the number of threads in a block and warp, and save the variables in the memory and restore them into the global memory.

4.3 Two Types of Data Transfer Methods

We transfer the input grammar from the host to the device before we start the membership test. After computation, we need to transfer the top cell of table, which is a set of variables deriving the input string, from device to host to verify the test. The data transfer between two devices often causes the degraded performance in GPU programming. It is crucial to reduce the data transfer time between the devices. We use the following two methods to improve the speed:

- **Page-locked Host Memory:** We generally allocate the data to the page-locked host memory. A page-locked buffer, also called pinned memory, save all data in physical memory. We can improve the speed of the data movement using page-locked host memory since this memory does not use paging.
- **Zero-copy Host Memory:** The zero-copy host memory enables GPUs to access host memory directly without transferring data to the device. In addition, GPUs can read and write data simultaneously in the host memory, which is not possible in a traditional PCI bus.

5 Experimental Results

We apply previous approaches to GPU. The details of the experimental platform are as follows: CPU is Core i3 3.10Ghz, RAM is 8GB, GPU is GTX 560 Ti and its memory is 1GB. The number of SM and SP are respectively 8 and 384. Shared Memory/SM is up to 48KB and L1 cache/SM is up to 512KB.

We use grammars by Petrov et al. [16] for our experiment. These grammars have been widely used for evaluating the performance of parsing with CFGs [2,9,20]. They suggested various methods such as splitting and merging variables for a high parsing performance. Because of splitting and merging rules, there are 1,043 variables and 1,725,570 binary rules for parsing in the resulting CFGs.

Since we only consider CFGs instead of PCFGs for the CYK algorithm, we ignore these splitting and merging variables in the experiment. We merge the same word class variables into a single variable. Therefore, we have 98 variables and 3,840 binary rules. We use Section 23 of the WSJ portion of the Penn Treebank [14] as our benchmark set.[1] The sequential version of the CYK algorithm was written in C. It requires $53,987$ ms per sentence. We compare the execution time of various implementations of the CYK algorithm in CUDA based on the thread mapping methods and different memory access patterns.

[1] In the benchmark set, an input string is a sentence, and the length of the input string is the number of words in the sentence.

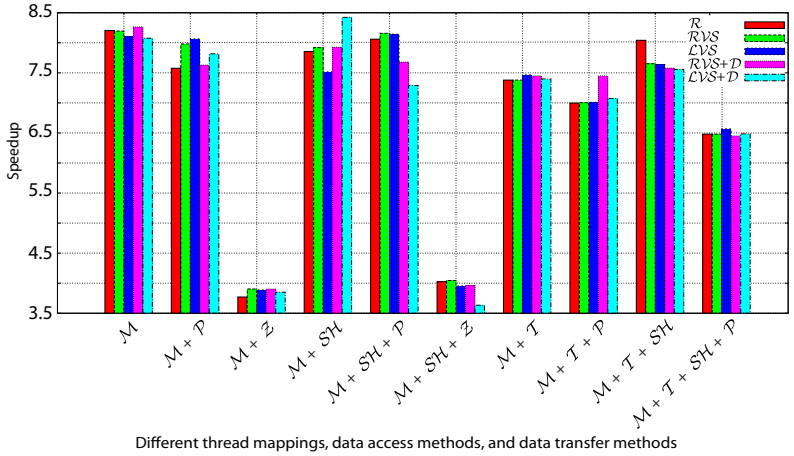

Fig. 3. Speedup of different implementations of a parallel CYK algorithm using different thread mapping methods, data access methods, and data transfer methods

Fig. 3 and Table 2 shows the speedup of the different parallel CYK algorithm implementations on a GTX 560 Ti.

- $\mathcal{M}$: Three thread mapping methods
- $\mathcal{R}$: Rule-based mapping
- $\mathcal{D}$: Deploying dummy rules to $\mathcal{RVS}$ and $\mathcal{LVS}$ mapping
- $\mathcal{P}$: Page-locked host memory
- $\mathcal{Z}$: Zero-copy host memory
- $\mathcal{SH}$: Storing data in shared memory
- $\mathcal{T}$: Placing grammar rules in texture memory

For each test, we randomly selected 1,000 sentences for the benchmark set and measure the runtime. We repeated this test 100 times and computed the average runtime for each case.

- **Five Mapping Methods:** We first implement rule-based, $\mathcal{LVS}$ and $\mathcal{RVS}$ mappings. We then add dummy rules to $\mathcal{LVS}$ and $\mathcal{RVS}$ mappings.
  1. $\mathcal{R}$: Rule-based mapping shows an 8.20× speedup.
  2. $\mathcal{RVS}$: The use of $\mathcal{RVS}$ shows an 8.18× speedup, which is similar to the result of rule-based mapping.
  3. $\mathcal{RVS}+\mathcal{D}$: Once we add dummies to $\mathcal{RVS}$, the size of the grammar increases by 46% to 5,632 compared with the original size of 3,840. Since the size of grammar increases, there might be a more frequent memory accesses in the kernel. However, we achieved a slightly improved performance of 8.26×, because of reducing thread divergence. Note that an advantage is achieved from all threads passing the first if-statement without any idle time.

4. $\mathcal{LVS}$: $\mathcal{LVS}$ shows a poorer performance than $\mathcal{R}$, with an 8.10× speedup. Since we sort the production rules according to the LHS variables, some rules that have different first RHS variables are processed in the same warp concurrently, which causes frequent thread divergences than $\mathcal{R}$.
5. $\mathcal{LVS} + \mathcal{D}$: When we add dummy rules, the grammar size increases by 21% to 4,630 compared with 3,840. The overhead becomes signification, so it shows only an 8.08× speedup, which is slower than $\mathcal{M}$.

Table 2. Speedups of different implementations of a parallel CYK algorithms by different thread mapping methods, data access methods, and data transfer methods

	$\mathcal{M}$	$\mathcal{M}+\mathcal{P}$	$\mathcal{M}+\mathcal{Z}$	$\mathcal{M}+\mathcal{SH}$	$\mathcal{M}+\mathcal{SH}+\mathcal{P}$	$\mathcal{M}+\mathcal{SH}+\mathcal{Z}$
$\mathcal{R}$	8.20	7.58	3.77	7.85	8.06	4.02
$\mathcal{RVS}$	8.19	7.98	3.90	7.92	8.16	4.05
$\mathcal{LVS}$	8.10	8.06	3.89	7.52	8.14	3.95
$\mathcal{RVS}+\mathcal{D}$	8.26	7.63	3.90	7.93	7.68	3.96
$\mathcal{LVS}+\mathcal{D}$	8.08	7.82	3.85	8.42	7.29	3.63

	$\mathcal{M}+\mathcal{S}$	$\mathcal{M}+\mathcal{T}+\mathcal{P}$	$\mathcal{M}+\mathcal{T}+\mathcal{SH}$	$\mathcal{M}+\mathcal{T}+\mathcal{SH}+\mathcal{P}$
$\mathcal{R}$	7.38	7.00	8.04	6.48
$\mathcal{RVS}$	7.38	7.00	7.65	6.48
$\mathcal{LVS}$	7.47	7.01	7.64	6.57
$\mathcal{RVS}+\mathcal{D}$	7.45	7.45	7.58	6.45
$\mathcal{LVS}+\mathcal{D}$	7.40	7.07	7.56	6.48

- **Page-Locked Host Memory:** Since our program uses a relatively small amount of memory, the page-locked host memory method shows a slower runtime compared with the memory allocation without a page-lock. This is because the paged-locked host memory has its own memory allocation function, which is slower than the traditional memory allocation function.
- **Zero-Copy Host Memory:** $\mathcal{M}+\mathcal{Z}$ and $\mathcal{M}+\mathcal{SH}+\mathcal{Z}$ show speedup of 4×, which is worse than $\mathcal{M}$, since the use of zero-copy host memory is very costly in our implementation as small amount of data are frequently moved.
- **Shared Memory:** In the case of shared memory, only $\mathcal{LVS}+\mathcal{D}+\mathcal{SH}$ shows a performance improvement. Since $\mathcal{LVS}+\mathcal{D}+\mathcal{SH}$ allocates shared memory according to the number of warps in each block, it can effectively utilize shared memory. This gives rise to a speedup of 8.42× at maximum. Note that the GTX 560 Ti has an L1 cache with a Fermi architecture [15]. We also observe that the other mappings have a shared memory table array in L1 cache and they also need more shared memory in proportion to the size of the variables and more access to the memory than those for $\mathcal{LVS}+\mathcal{D}$ mapping. The other mappings using shared memory therefore degrade the performance.
- **Texture Memory:** For texture memory, we observe a slower speed than the case without texture mapping for two reasons: First, the GPUs must fetch data from texture memory by calling a special fetch operation, which delays the process overall. Second, the texture memory is optimized for 2D data [18], whereas our data is a 1D string.

We also compared the implementation of a previous study by Yi et al [21]. Since their implementation is for parsing PCFG, it requires an additional operation to calculate the probability, atomic operation, and so on. We therefore cannot compare their implementation directly. The speedup of the fastest implementation of their work is 4.37×. This result is worse than our result 8.42×. This means that our result is more appropriate for a CFG membership test.

6 Conclusions

We explored a design for parallelizing the CYK algorithm. We analyzed different methods for thread mapping, data access using various types of memories and data transfer based on the memory design concepts. We then compared the different implementations of the CYK algorithm on a GTX560Ti. Our contributions can be summarized as follows:

- We evaluated various implementations of CYK on a GPU
- We utilized a memory access pattern in a warp using shared memory

The fastest implementation of the algorithm when using a GPUs is from $\mathcal{LVS} + \mathcal{D} + \mathcal{SH}$ mapping when the number of rules is relatively small (in our test case, it was almost 8k), i.e., left-variable sorting while deploying dummy rules with shared memory. This implementation is 8.42× faster than the sequential C version. However, the experimental results showed that except for $\mathcal{LVS} + \mathcal{D}$ mapping, using shared memory is slower than using global memory because the data are already in the L1 cache on the GTX560Ti using the 3,841 production rules of the benchmark grammar. Using page-locked and zero-copy host memory results in more overhead, and compared to its benefit, it worsens the performance. We believe that these observations will be helpful for designing a fast parallel CYK algorithm for use on GPUs.

References

1. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling (1972)
2. Bodenstab, N., Dunlop, A., Hall, K., Roark, B.: Beam-width prediction for efficient context-free parsing. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp. 440–449 (2011)
3. Bordim, J.L., Ito, Y., Nakano, K.: Accelerating the CKY parsing using fPGAs. In: Sahni, S.K., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 41–51. Springer, Heidelberg (2002)
4. Cai, L., Malmberg, R.L., Wu, Y.: Stochastic modeling of RNA pseudoknotted structures: a grammatical approach. Bioinformatics, 66–73 (2003)
5. Chomsky, N.: On certain formal properties of grammars. Information and Control, 137–167 (1959)
6. Cocke, J.: Programming languages and their compilers: Preliminary notes (1969)
7. D'Agostino, D., Clematis, A., Decherchi, S., Rocchia, W., Milanesi, L., Merelli, I.: Cuda accelerated molecular surface generation. Concurrency and Computation: Practice and Experience 26(10), 1819–1831 (2014)

8. Dunlop, A., Bodenstab, N., Roark, B.: Efficient matrix-encoded grammars and low latency parallelization strategies for CYK. In: Proceedings of the 12th International Conference on Parsing Technologies, pp. 163–174 (2011)
9. Foster, J.: "cba to check the spelling" investigating parser performance on discussion forum posts. In: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 381–384 (2010)
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (1979)
11. Johnson, M.: Parsing in parallel on multiple cores and GPUs. In: Proceedings of the Australasian Language Technology Association Workshop 2011, pp. 29–37 (2011)
12. Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory (1965)
13. Khronos OpenCL Working Group. The OpenCL Specification, version 1.0.29 (2008), http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf
14. Marcus, M.P., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: The Penn Treebank. Computational Linguistics 19(2), 313–330 (1993)
15. Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, Nvidia Corporation (2009)
16. Petrov, S., Barrett, L., Thibaux, R., Klein, D.: Learning accurate, compact, and interpretable tree annotation. In: Proceedings of the 21st International Conference on Computational Linguistics, pp. 433–440 (2006)
17. Sakakibara, Y.: Learning context-free grammars using tabular representations. Pattern Recognition 38(9), 1372–1383 (2005)
18. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional (2010)
19. Takashi, N., Kentaro, T., Taura, K., Tsujii, J.: A parallel CKY parsing algorithm on large-scale distributed-memory parallel machines. In: Proceedings of the 5th Pacific Association For Computational Lingustics, pp. 223–231 (1997)
20. Weese, J., Ganitkevitch, J., Callison-Burch, C., Post, M., Lopez, A.: Joshua 3.0: syntax-based machine translation with the thrax grammar extractor. In: Proceedings of the 6th Workshop on Statistical Machine Translation, pp. 478–484 (2011)
21. Yi, Y., Lai, C.-Y., Petrov, S.: Efficient parallel CKY parsing using GPUs. Journal of Logic and Computation 24(2), 375–393 (2014)
22. Younger, D.H.: Recognition and parsing of context-free languages in time n^3. Information and Control 10, 189–208 (1967)
23. Vu, V., Cats, G., Wolters, L.: Graphics processing unit optimizations for the dynamics of the HIRLAM weather forecast model. Concurrency and Computation: Practice and Experience 25(10), 1376–1393 (2013)