Project Report
CYK Algorithm on GPUs
Prateek Roy, Akshay Satam

1. Problem description:

We implemented serial CYK parsing algorithms (mentioned in paper[1]) and also both the parallel approaches of CYK Parsing algorithm using GPUs (mentioned in paper[1]) in order to reduce the computation time. The CKY dynamic programming is used to find the most likely parse tree for a given sentence of length n in O(G n^3) time. While often ignored, the grammar constant G typically dominates the runtime in practice. This is because grammars with high accuracy have thousands of non terminal symbols and millions of context free rules, while most sentences are pretty small having around 20 words. Because of these costly computations, serially generating the parse-tree takes a huge amount of time. This complexity of grammar rules and symbols led us to think about parallelizing the CYK algorithm on GPUs. The application of the generated parse trees are numerous starting from machine translation, question answering and information extraction.

2. Prior Work :

The parallel parsers in past work are implemented on multi-core systems, where the limited parallelization possibilities provided by the systems restrict the speedups that can be achieved. In contrast, the parallel parser suggested by the paper is implemented on a many core system with an abundant number of threads and processors to parallelize upon. Previous work was based on agenda based parsers (maintain a queue of prioritized intermediate results and iteratively refine and combine these until the whole sentence is processed), thus limiting parallelism because only a small number of intermediate results can be handled simultaneously. However the paper suggests chart based parsing. There were other hardware approaches to speedup the algorithm, but those suffers from insufficient memory or logic elements and limits the number of rules in the grammar to 2,048 and the number of non-terminal symbols. Their approach thus cannot be applied to real-world, state-of-the art grammars. There was another proposal of a parallel CYK parser on distributed-memory parallel machine consisting of 256 cores, where synchronization over cores is implemented by message passing, preventing them from parallelizing over rules and nonterminal symbols. The paper suggests to parallelize over rules and nonterminal symbols, as well as cells, and address the load imbalance

problem by introducing virtual symbols as discussed in the
paper. There were also numerous orthogonal approaches which can increase speedup but introduces search errors.

3. Original goal(s) / deliverable(s):

Project Implementation:

- An implementation of the CYK serial algorithm based on the pseudocode (Fig 3 in paper[1])
- An implementation of the CYK Parser parallel algorithm on GPU, based on the approach discussed in the paper[1] (thread based mapping, block based mapping (Fig. 6 and 7)).

Project analysis:

- Analyzing the thread based parallel, block based parallel and the serial algorithms mentioned in the paper.

4. Summary of accomplishments:

- Implemented the CYK serial algorithm.
- Implemented the Parallel CYK algorithm using two approaches : Thread based mapping and Block based mapping
- Analysis of serial algorithm and both thread based as well as block based mapping approach.

5. Implementation details:

Serial CYK Algorithm

- This is a dynamic programming algorithm which uses two 3D matrix - Scores and Backpointer.
- Scores matrix maintains the score of the Constituency Parse Tree while Backpointer matrix maintains the backpointers, which helps in generating the Constituency Parse Tree.
- Initially all the diagonal elements of the scores matrix are filled. These diagonal elements correspond to the actual words in the sentence. Hence, all the lexicons rules which produce these words are examined and the score matrix is updated in location corresponding to the symbol and the index of the word. Once this is done, all the unary rules are checked to see if they produce the symbols in above steps. If yes, then locations corresponding to the producing symbol and the word index is updated.
- Then we take a batch of words from the beginning of the sentence, starting with batch-size 2 and keep increasing the batch-size. For each batch-size, we try to break the batch at different possible positions and then see if the two symbols in each partition can be obtained from some binary rule. If yes,

then the score is updated in corresponding location.

Please find the formula for the DP:

For every binary rule of the form: sym -> lsym rsym

Score [i,j] = argmax { score[i][k][lsym] + score[k+1][j][rsym] + probability_score[rule]}

```
Algorithm: parse(sen, lex, gr)
Input: sen /* the input sentence */
       lex /* the lexicon */
       gr /* the grammar */
Output: tree /* the most probable parse tree */

1  scores[][][] = initScores();
2  nWords = readSentence(sen);
3  lexiconScores(scores, sen, nWords, lex);
4  for length = 2 to nWords
5      binaryRelax(scores, nWords, length, gr);
6      unaryRelax(scores, nWords, length, gr);
7  tree = backtrackBestParseTree(scores);
8  return tree;
```

```
Algorithm: binaryRelax(scores, nWords, length, gr)
Input: scores /* the 3-dimensional scores */
       nWords /* the number of total words */
       length /* the current span */
       gr /* the grammar */
Output: None

1   for start = 0 to nWords − length
2       end = start + length;
3       foreach symbol ∈ gr
4           max = FLOAT_MIN;
5           foreach rule r per symbol // defined by gr
6               // r is "symbol ⇒ l_sym r_sym"
7               for split = start + 1 to end − 1
8                   // calculate score
9                   lscore = scores[start][split][l_sym];
10                  rscore = scores[split][end][r_sym];
11                  score = rule_score + lscore + rscore;
12                  // maximum reduction
13                  if score > max
14                      max = score;
15          scores[start][end][symbol] = max;
```

Pseudocode for Serial CYK Algorithm

1. **CYK Algorithm on GPU - Thread based**

```
Algorithm: threadBasedRuleBR(scores, nWords, length, gr)
Input: scores /* the 3-dimensional scores */
       nWords /* the number of total words */
       length /* the current span */
       gr /* the grammar */
Output: None

1   for start = 0 to nWords − length in parallel
2       end = start + length;
3       foreach rule r ∈ gr in parallel
4           __shared__ int sh_max[NUM_SYMBOL] =
                                    FLOAT_MIN;
5           // r is "symbol ⇒ l_sym r_sym"
6           for split = start + 1 to end − 1
7               // calculate score
8               lscore = scores[start][split][l_sym];
9               rscore = scores[split][end][r_sym];
10              score = rule_score + lscore + rscore;
11              // local maximum reduction
12              if score > local_max
13                  local_max = score;
14          atomicMax(&sh_max[symbol], local_max);
15  // global maximum reduction
16  foreach symbol ∈ gr in parallel
17      atomicMax(&scores[start][end][symbol],
                                    sh_max[symbol]);
```

Thread Based Mapping

With millions of grammar rules and symbols, we can map either rules or symbols to a thread. However if we map a symbol to a thread, then it fails to provide enough parallelism to fully utilize the massive number of threads in GPU and can also lead to load imbalance because each symbol can have varying number of rules associated with it. A thread with a symbol having millions of rules can be overloaded where a thread with a symbol having hundred rule is underloaded. Since threads in the same warp execute in SIMT fashion, this load imbalance among threads results in divergent branches, degrading the performance significantly. It is therefore

advantageous to map rules rather than symbols to threads.

Now if we map rules to threads, then there will be only one loop that iterates over all rules in the grammar and the loop can be parallelized by executing each rule in a thread of GPU. As the rules in the grammar are present in very large number (in millions), this mapping can utilize the GPU and provide sufficient parallelism without running into load imbalance.

Disadvantages : Unfortunately, thread-based mapping has a major drawback. Since each rule is mapped to a different thread, threads for rules with the same parent symbol need to be synchronized in order to avoid write conflicts. In this mapping, the synchronization can be done only through atomic operations, which can be costly. When we call atomic operations on shared memory, shared variables need to be declared for all symbols. This is necessary because in thread-based mapping threads in the same thread block can have different parent symbols.

2. CYK Algorithm on GPU - Block based

```
Algorithm: blockBasedRuleBR(scores, nWords, length, gr)
Input: scores /* the 3-dimensional scores */
       nWords /* the number of total words */
       length /* the current span */
       gr /* the grammar */
Output: None

1   for start = 0 to nWords − length in parallel
2       end = start + length;
3       foreach symbol ∈ gr in parallel
4           __shared__ int sh_max = FLOAT_MIN;
5           foreach rule r per symbol in parallel
6               // r is "symbol ⇒ l_sym r_sym"
7               for split = start + 1 to end − 1
8                   // calculate score
9                   lscore = scores[start][split][l_sym];
10                  rscore = scores[split][end][r_sym];
11                  score = rule_score + lscore + rscore;
12                  // local maximum reduction
13                  if score > local_max
14                      local_max = score;
15              atomicMax(&sh_max, local_max);
16  // global maximum reduction
17  foreach symbol ∈ gr in parallel
18      atomicMax(&scores[start][end][symbol], sh_max);
```

We will be exploiting two levels of granularity of GPU architecture - thread and thread blocks. We already saw thread based mapping in last section. Now lets look at block based mapping. In this approach, we map a symbol to a thread block and each rules associated with that symbol to each thread inside the same thread block. This mapping creates a balanced load because an streaming multiprocessor(SM) can execute any available thread block independently of other thread blocks, instead of waiting for other SMs to complete. For example, suppose a thread block is associated with a symbol which has very few rules, so it completes first, the the SM processing that thread block can move to next block which corresponds to another symbol.

Advantages:
- It allows synchronization without costly atomic operations unlike the thread based approach. All threads in a thread block have the same parent symbol, and therefore only one shared variable per thread block is needed for the parent symbol.

- We can quickly skip over certain symbols. For example, the pre-terminal symbols can only cover spans of length 1 (NOUN -> Houston). In block based mapping only one thread needs to check and determine if a symbol is pre-terminal and it can skip it whereas in thread based mapping, every thread in the thread block needs to perform the check which is unnecessary.

6. Experimental Results:

Experimental Setup:
We used Comet Servers (comet.sdsc.xsede.org) for our experimental setup for running cuda code on GPUs available on comet.

To run the serial version of the CYK algorithm, follow steps:
- cd Serial
- Module load cuda
- Make
- ./run

To run the Thread Based and Block Based approaches enter into the proper folder and run the above commands:

- Cd Thread Based
- Cd Block Based
- Repeat above steps

Machine configuration while running Thread Based and Block Based was

srun --partition=gpu --nodes=1 --ntasks-per-node=6 --gres=gpu:4 -t 00:30:00 --pty --wait=0 --export=ALL /bin/bash

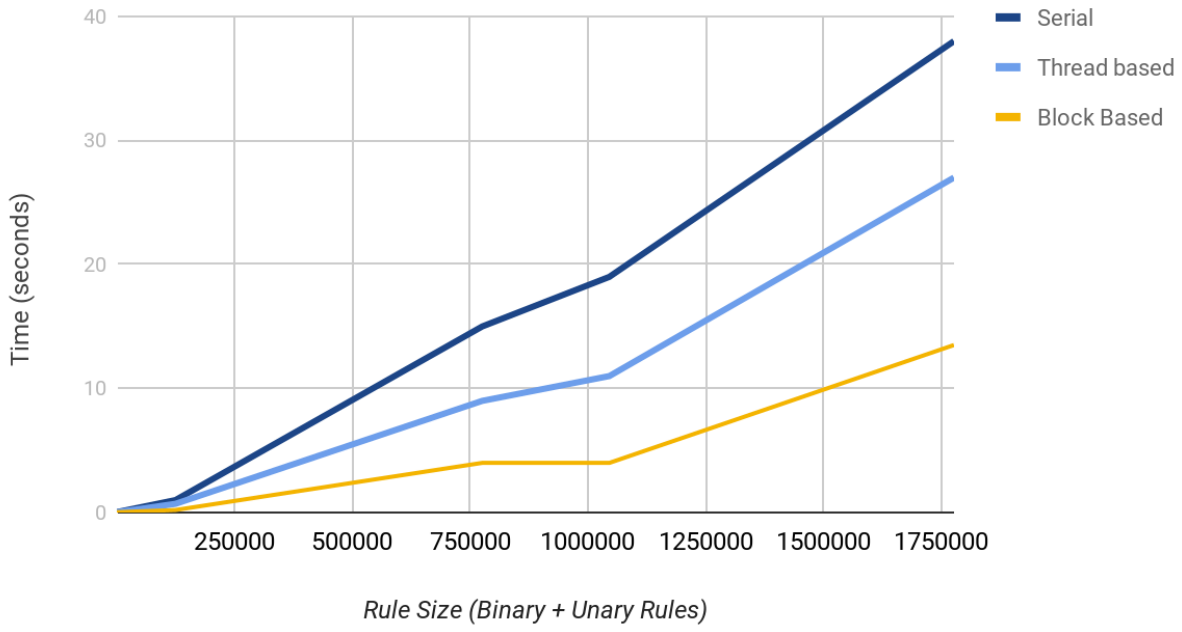
1 Node with 6 tasks per node and total of 4 GPUs.

Experiment Results:

We used the publicly available PetrovParser to get grammar. The grammar has **1777742 rules** which includes both unary and binary rules. Number of **symbols present are 106**. And we test our CYK Parser with the sentence "**Book the flight through Houston**". We carried this experiment for all algorithms (serial, thread based CUDA, block based CUDA) which we implemented for different grammars rules size as mentioned in the below table.
The grammar is present in **preProcessedGrammar.txt** and **preProcessedLexicons.txt** files.

Rule Size	Symbol Size	Serial	Thread Based	Block Based
54	14	0.03 milliseconds	0.02 milliseconds	0.02 milliseconds
124547	60	1 seconds	0.7 seconds	0.2 seconds
776066	61	15 seconds	9 seconds	4 seconds
1046075	61	19 seconds	11 seconds	4 seconds
1777742	106	38 seconds	27 seconds	13.5 seconds

Time Plot

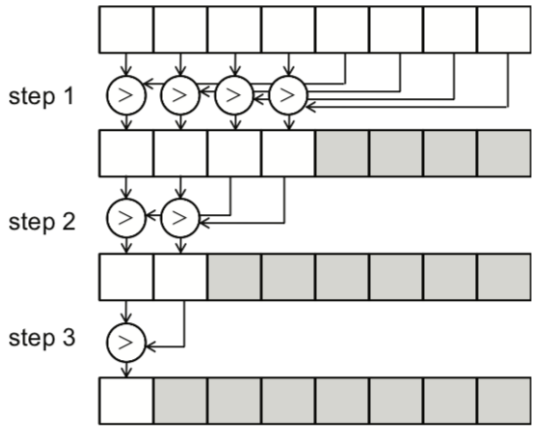


Analysis of results:

As we can see from the above table, the time taken by the serial algorithm is the most. The thread based approach has considerably less improvement in performance because the atomic operations on shared memory is a bottleneck. Block based mapping is the fastest as there is no bottleneck for shared memory and also there is no work overload. Due to these factors, Block based mapping is preferred in CYK algorithm and our experimental results justify it.

7. Future Work:

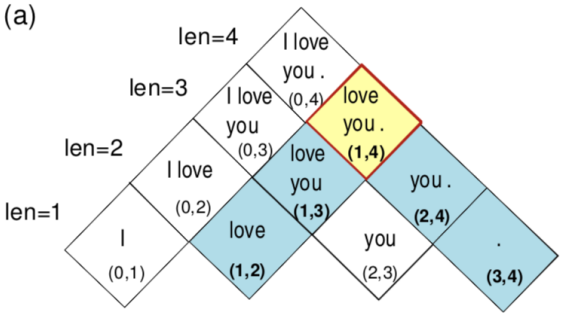
Parallel Reduction : This approach can be applied to block based mapping only and not to thread based mapping. Each thread in the thread block computes its score in the shared array and performs parallel reduction with binary tree order as shown in the figure (from leaves to root). While implementing the parallel reduction, syncthreads() needs to be called at the end of each step to ensure that every thread can read the updated value of the previous step. It is more faster than serial algorithm but it takes much more shared memory.



Parallel reduction between threads in same thread blocks.

Reducing Global Memory Access : If we look at the memory access pattern in the below figure, we can see that the access occur at specific areas for each cell. For example, the accesses are restricted within the shaded cells when the cell for evaluation is (1,4). We can make two arrays (sh scores Left and sh scores Right) that keeps track of the scores of children in the shaded areas. These arrays can easily fit into shared memory because there are only about 1000 unique symbols in our grammar and the sentence length are bounded. For each unique left/right child symbol, we need to load the score from global scores matrix to shared scores Left and shared scores Right matrix once through a global memory access. Thus, the number of global memory access will be reduced when multiple rules share the same child symbols. The psuedocode is given in below image.

(a)



(b)

for $split = start + 1$ **to** $end - 1$
$lscore = scores[start][split][l\_sym]$;
$rscore = scores[split][end][r\_sym]$;
$score = rule\_score + lscore + rscore$;

for $k = 1$ **to** $len - 1$
$lscore = sh\_scores\_L[k][unique\_l\_sym]$;
$rscore = sh\_scores\_R[k][unique\_r\_sym]$;
$score = rule\_score + lscore + rscore$;

Another way of reducing global memory access is to use texture memory. Texture memory can be used for caching but it is initialized from Host side and thus has high overhead. Scores matrix can be moved to texture memory as it is frequently read to access the scores of the children. However after every iteration of binaryRelax function, the array needs to be updated by calling a binding API which is costly operation. Thus we can change the structure of scores matrix in order to solve this problem. Instead of using scores[start][end][symbol], we can change the scores matrix to scores[length][start][symbol]. With this array we need to bind the array upto size(length -1) rather than entire array.

8. References

1. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37628.pdf
2. http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/pcfgs.pdf
3. http://www.cse.unsw.edu.au/~cs9414/Intro/notes/nlp/grampars.html
4. https://gist.github.com/xiaohan2012/b9e3ab0ac5d23362bf33
5. https://www.usna.edu/Users/cs/nchamber/courses/nlp/f13/labs/cky-pseudo.html
6. https://www.usna.edu/Users/cs/nchamber/courses/nlp/f13/labs/lab6.html