Efficient Parallel CKY Parsing on GPUs

Youngmin Yi[1] **Chao-Yue Lai**[2] **Slav Petrov**[3] **Kurt Keutzer**[4]

[1]University of Seoul
Seoul, Korea
ymyi@uos.ac.kr

[2,4]University of California, Berkeley
Berkeley, CA, USA
{colbylai,keutzer}
@eecs.berkeley.edu

[3]Google Research
New York, NY, USA
slav@google.com

Abstract

Low-latency solutions for syntactic parsing are needed if parsing is to become an integral part of user-facing natural language applications. Unfortunately, most state-of-the-art constituency parsers employ large probabilistic context-free grammars for disambiguation, which renders them impractical for real-time use. Meanwhile, Graphics Processor Units (GPUs) have become widely available, offering the opportunity to alleviate this bottleneck by exploiting the fine-grained data parallelism found in the CKY algorithm. In this paper, we explore the design space of parallelizing the dynamic programming computations carried out by the CKY algorithm. We use the Compute Unified Device Architecture (CUDA) programming model to reimplement a state-of-the-art parser, and compare its performance on two recent GPUs with different architectural features. Our best results show a 26-fold speedup compared to a sequential C implementation.

1 Introduction

Syntactic parsing of natural language is the task of analyzing the grammatical structure of sentences and predicting their most likely parse trees (see Figure 1). These parse trees can then be used in many ways to enable natural language processing applications like machine translation, question answering, and information extraction. Most syntactic constituency parsers employ a weighted context-free grammar (CFG), that is learned from a treebank. The CKY dynamic programming algorithm (Cocke and Schwartz, 1970; Kasami, 1965; Younger, 1967) is then be 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 (Collins, 1999; Charniak, 2000; Petrov et al., 2006) have thousands of nonterminal symbols and millions of context-free rules, while most sentences have on average only about $n = 20$ words.

Meanwhile, we have entered a manycore computing era, where the number of processing cores in computer systems doubles every second year, while the clock frequency has converged somewhere around 3 GHz (Asanovic et al., 2006). This opens up new opportunities for increasing the speed of parsers. We present a general approach for parallelizing the CKY algorithm that can handle arbitrary context-free grammars (Section 2). We make no assumptions about the size of the grammar and we demonstrate the efficacy of our approach by implementing a decoder for the state-of-the-art latent variable grammars of Petrov et al. (2006) (a.k.a. Berkeley Parser) on a Graphics Processor Unit (GPU).

We first present an overview of the general architecture of GPUs and the efficient synchronization provided by the Compute Unified Device Architecture (CUDA (Nickolls et al., 2008)) programming model (Section 3). We then discuss how the hundreds of cores available on a GPU can enable a fine-grained parallel execution of the CKY algorithm. We explore the design space with different thread mappings onto the GPU and discuss how the various synchronization methods might be applied in this context (Section 4). Key to our approach is the observation that the computation needs to be parallelized over grammar rules rather than chart cells. While this might have been difficult to do with previous parallel computing architectures, the CUDA model provides us with

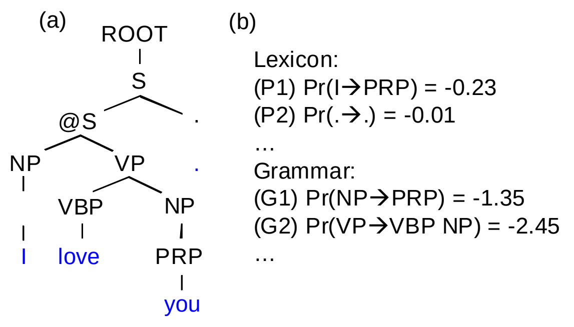


Figure 1: An example of a parse tree for the sentence "I love you ."

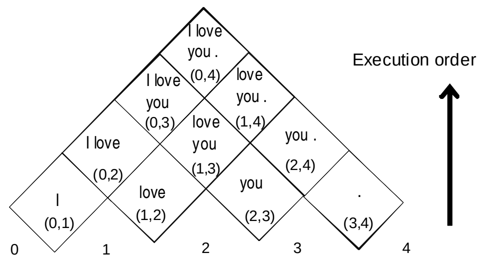


Figure 2: The chart that visualizes the bottom-up process of CKY parsing for the sentence "I love you ."

fine-grained parallelism and synchronization options that make this possible.

We empirically evaluate the various parallel implementations on two NVIDIA GPUs (GTX480 and GTX285) in Section 5. We observe that some parallelization options are architecture dependent, emphasizing that a thorough understanding of the programming model and the underlying hardware is needed to achieve good results. Our implementation on NVIDIA's GTX480 using CUDA results in a 26-fold speedup compared to the original sequential C implementation. On the GTX285 GPU we obtain a 14-fold speedup.

Parallelizing natural language parsers has been studied previously (see Section 6), however, previous work has focused on scenarios where only a limited level of coarse-grained parallelism could be utilized, or the underlying hardware required unrealistic restrictions on the size of the context-free grammar. To the best of our knowledge, this is the first GPU-based parallel syntactic parser using a state-of-the-art grammar.

2 Natural Language Parsing

While we assume a basic familiarity with probabilistic CKY parsing, in this section we briefly review the CKY dynamic programming algorithm and the Viterbi algorithm for extracting the highest scoring path through the dynamic program.

2.1 Context-Free Grammars

In this work we focus our attention on constituency parsing and assume that a weighted CFG is available to us. In our experiments we will use a probabilistic latent variable CFG (Petrov et al., 2006). However, our algorithms can be used with any weighted CFG, including discriminative ones, such as the ones in Petrov and Klein (2007a) and Finkel et al. (2008).[1] The grammars in our experiments have on the order of thousands of nonterminals and millions of productions.

Figure 1(a) shows a constituency parse tree. Leaf nodes in the parse tree, also called terminal nodes, correspond to words in the language. Preterminals correspond to part-of-speech tags, while the other nonterminals correspond to phrasal categories. For ease of exposition, we will say that terminal productions are part of a lexicon. For example, (L1) in Figure 1(b) is a lexical rule providing a score (of -0.23) for mapping the word "I" to the symbol "PRP." We assume that the grammar has been binarized and contains only *unary* and *binary* productions. We refer to the application of grammar rules as *unary/binary relaxations*.

2.2 Sequential CKY Parsing

The CKY algorithm is an exhaustive bottom-up algorithm that uses dynamic programming to incrementally build up larger tree structures. To keep track of the scores of these structures, a chart indexed by the start and end positions and the symbol under consideration is used: *scores*[*start*][*end*][*symbol*] (see also Figure 2). After initializing the preterminal level of the chart with the part-of-speech scores from the lexicon, the algorithm continues by repeatedly applying all binary and unary rules in order to build up larger spans (pseudocode is given in Figure 3). To reconstruct the highest scoring parse tree we perform a top-down search. We found this to be more efficient than keeping backpointers.[2]

One should also note that many real-world applications benefit from, or even expect n-best lists of possible parse trees. Using the lazy evaluation algorithm of Huang and Chiang (2005) the extrac-

[1]For feature-rich discriminative models a trivially parallelizable pass can be used to pre-compute the rule-potentials.

[2]This observation is due to Dan Klein, *p.c.*

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

$scores$[][][] = initScores();
$nWords$ = readSentence($sen$);
lexiconScores($scores, sen, nWords, lex$);
for $length = 2$ to $nWords$
    binaryRelax($scores, nWords, length, gr$);
    unaryRelax($scores, nWords, length, gr$);
$tree$ = backtrackBestParseTree($scores$);
return $tree$;
```

Figure 3: Pseudocode for CKY parsing.

```
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

for $start = 0$ to $nWords - length$
    $end = start + length$;
    foreach $symbol \in gr$
        $max$ = FLOAT_MIN;
        foreach rule $r$ per $symbol$ // defined by gr
            // r is "symbol ⇒ l_sym r_sym"
            for $split = start + 1$ to $end - 1$
                // calculate score
                $lscore = scores[start][split][l\_sym]$;
                $rscore = scores[split][end][r\_sym]$;
                $score = rule\_score + lscore + rscore$;
                // maximum reduction
                if $score > max$
                    $max = score$;
        $scores[start][end][symbol] = max$;
```

Figure 4: Binary relaxations in CKY parsing.

tion of an n-best list can be done with very little overhead after running a slightly modified version of the CKY algorithm. Our parallel CKY algorithm could still be used in that scenario.

3 GPUs and CUDA

Graphics Processor Units (GPUs) were originally designed for processing graphics applications, where millions of operations can be executed in parallel. In order to increase the efficiency by exploiting this parallelism, typical GPUs (Lindholm et al., 2008) have hundreds of processing cores. For example, the NVIDIA GTX480 GPU has 480 processing cores called *stream processors* (SP). The processing cores are organized hierarchically as shown in Figure 5: A group of SPs makes up a *streaming multiprocessor* (SM). A number of SMs form a single graphics device. The GTX480, for example, contains 15 SMs, with 32 SPs in each SM, resulting in the total of 480 SPs. Despite this high number of processors, it should be emphasized that simply using a GPU, without understanding the programming model and the underlying hardware architecture, does not guarantee high performance.

3.1 Compute Unified Device Architecture

Recently, Nickolls et al. (2008) introduced the Compute Unified Device Architecture (CUDA). It allows programmers to utilize GPUs to accelerate applications in domains other than graphics. CUDA is essentially the programming language C with extensions for thread execution and GPU-specific memory access and control. A CUDA *thread* is executed on an SP and a group of threads (called a *thread block*) is executed on an SM. CUDA enables the acceleration of a wide range of applications in various domains by executing a number of threads and thread blocks in parallel, which are specified by the programmer. Its popularity has grown rapidly because it provides a convenient API for parallel programming. In order to better utilize the massive parallelism in the GPU, it is typical to have hundreds of threads in a thread block and have hundreds or even thousands of thread blocks launched for a single *kernel*: a data-parallel function that is executed by a number of threads on the GPU.

3.2 Single Instruction Multiple Threads

One of the most important features of the GPU architecture is commonly known as Single Instruction Multiple Threads (SIMT). SIMT means that threads are executed in bundles (called *warps*), to amortize the implementation cost for a large number of processing cores. In typical GPUs, a warp consists of 32 threads that share the units for instruction fetching and execution. Thus, a thread cannot advance to the next instruction if other threads in the same warp have not yet completed their own execution. On the other hand, threads in different warps are truly independent: they can be scheduled and executed in any order. Inside a warp, if some threads follow different execution paths than others, the execution of the threads with different paths is serialized. This can happen for example in if-then-else structures and loop constructs where the branch condition is based on thread indices. This is called a *divergent*

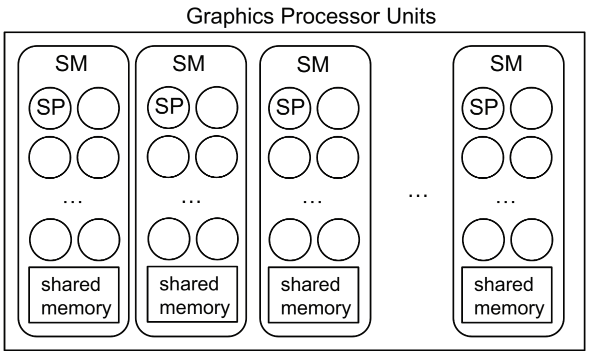


Figure 5: The hierarchical organization of GPUs.

branch and should be avoided as much as possible when designing parallel algorithms and mapping applications onto a GPU. In the programming model of CUDA, one or more warps are implicitly grouped into thread blocks. Different thread blocks are mapped to SMs and can be executed independently of one another.

3.3 Shared Memory

Generally speaking, manycore architectures (like GPUs) have more ALUs in place of on-chip caches, making arithmetic operations relatively cheaper and global memory accesses relatively more expensive. Thus, to achieve good performance, it is important to increase the ratio of Compute to Global Memory Access (CGMA) (Kirk and Hwu, 2010), which can be done in part by cleverly utilizing the different types of shared on-chip memory in each SM.

Threads in a thread block are mapped onto the same SM and can cooperate with one another by sharing data through the on-chip *shared memory* of the SM (shown in Figure 5). This shared memory has two orders of magnitude less latency than the off-chip *global memory*, but is very small (16KB to 64KB, depending on the architecture). CUDA therefore provides the programmer with the flexibility (and burden) to explicitly manage shared memory (i.e., loading a value from global memory and storing it).

Additionally, GPUs also have so called *texture memory* and *constant memory*. Texture memory can be written only from the host CPU, but provides caching and is shared across different SMs. Hence it is often used for storing frequently accessed read-only data. Constant memory is very small and as its name suggests is only appropriate for storing constants used across thread blocks.

3.4 Synchronization

CUDA provides a set of APIs for thread synchronization. When threads perform a reduction, or need to access a single variable in a mutually exclusive way, *atomic operations* are used. Atomic operation APIs take as arguments the memory location (i.e., pointer of the variable to be reduced) and the value. However, atomic operations on global memory can be very costly, as they need to serialize a potentially large number of threads in the kernel. To reduce this overhead, one usually applies atomic operations first to variables declared in the shared memory of each thread block. After these reductions have completed another set of atomic operations is done.

In addition, CUDA provides an API (*__syncthreads()*) to realize a barrier synchronization between threads in the same thread block. This API forces each thread to wait until all threads in the block have reached the calling line. Note that there is no API for the barrier synchronization between all threads in a kernel. Since a return from a kernel accomplishes a global barrier synchronization, one can use separate kernels when a global barrier synchronization is needed.

4 Parallel CKY Parsing on GPUs

The dynamic programming loops of the CKY algorithm provide various types of parallelism. While the loop in Figure 3 cannot be parallelized due to dependencies between iterations, all four loops in Figure 4 could in principle be parallelized. In this section, we discuss the different design choices and strategies for parallelizing the binary relaxation step that accounts for the bulk of the overall execution time of the CKY algorithm.

4.1 Thread Mapping

The essential step in designing applications on a parallel platform is to determine which execution entity in the parallel algorithm should be mapped to the underlying parallel hardware thread in the platform. For a CKY parser with millions of grammar rules and thousands of symbols, one can map either rules or symbols to threads. At first sight it might appear that mapping chart cells or symbols to threads is a natural choice, as it is equivalent to executing the first loop in Figure 4 in parallel. However, if we map a symbol to a thread, then it not only fails to provide enough parallelism to fully utilize the massive number of threads in

```
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 $r \in gr$ 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 $symbol \in gr$ in parallel
17      atomicMax(&$scores[start][end][symbol]$,
                                          $sh\_max[symbol]$);
```

Figure 6: Thread-based parallel CKY parsing.

```
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 $symbol \in gr$ 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 $symbol \in gr$ in parallel
18      atomicMax(&$scores[start][end][symbol]$, $sh\_max$);
```

Figure 7: Block-based parallel CKY parsing.

GPUs,[3] but it can also suffer from load imbalance due to the fact that each symbol has a varying number of rules associated with it. 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.

4.1.1 Thread-Based Mapping

If we map rules to threads, the nested loops in line 3 and line 5 of Figure 4 become one flat loop that iterates over all rules in the grammar and the loop can be executed in parallel as shown in line 3 of Figure 6. Since the grammar we use has about one million rules, this mapping provides sufficient parallelism to fully utilize the GPU, without running into load imbalance issues. We call this mapping *thread-based mapping*.

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 (shown in line 14 and line 17 of Figure 6), which can be costly.

[3] $\#threads/warp \times max(\#warps)/SM \times \#SM = 32 \times 48 \times 15 =$ 23,040 threads in the GTX480.

4.1.2 Block-Based Mapping

Another mapping can be obtained by exploiting the two levels of granularity in the GPU architecture: threads and thread blocks. We can map each symbol to a thread block, and the rules associated with each symbol to threads in the respective thread block. This mapping creates a balanced load because an SM can execute any available thread block independently of other thread blocks, instead of waiting for other SMs to complete. For example, when the first SM completes the computation of a thread block (because the associated symbol has few rules), it can proceed to the next available thread block (which corresponds to another symbol). This corresponds to mapping each iteration of the loop in line 3 of Figure 4 to thread blocks and the loop in line 5 to threads. We call this mapping *block-based mapping* and provide pseudocode in Figure 7. The main advantage of this mapping is that it allows synchronization without using costly atomic operations.

Another advantage of the block-based mapping is that we can quickly skip over certain symbols. For example, the preterminal symbols (i.e. part-of-speech tags), can only cover spans of length 1 (i.e. single words). In block-based mapping, only one thread needs to check and determine if a symbol is a preterminal and can be skipped. In contrast, in thread-based mapping, every thread in the

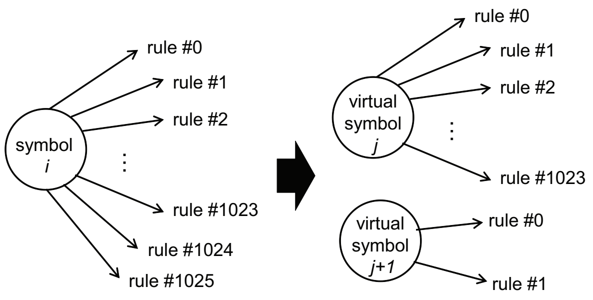


Figure 8: Creating virtual symbols whenever a symbol has too many rules.

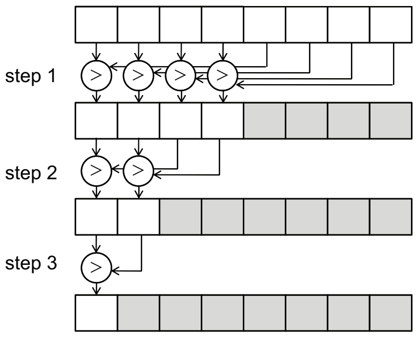


Figure 9: Parallel reduction on shared memory between threads in the same thread block with 8 threads.

thread block needs to perform the check. Since this check involves a global memory access, it is costly. Minimizing the number of global memory accesses is key to the performance of parallel algorithms on GPUs.

A challenging aspect of the block-based mapping comes from the fact that the number of rules per symbol can exceed the maximum number of threads per thread block (1,024 or 512 depending on the GPU architecture). To circumvent this limitation, we introduce virtual symbols, which host different partitions of the original rules, as shown in Figure 8. Introducing virtual symbols does not increase the complexity of the algorithm, because virtual symbols only exist until we perform the maximum reductions, at which point they are converted to the original symbols.

4.2 Span-Level Parallelism

Another level of parallelism, which is orthogonal to the previously discussed mappings, is present in the first loop in Figure 4. Spans in the same level in the chart (see Figure 2), are independent of each other and can hence be executed in parallel by mapping them to thread blocks (line 1 of Figure 6 and Figure 7). Since CUDA provides up to three-dimensional (x, y, z) indexing of thread blocks, this can be easily accomplished: we create two-dimensional grids whose X axis corresponds to symbols in block-based mapping or simply a group of rules in thread-based mapping, and the Y axis corresponds to the spans.

4.3 Thread Synchronization

Thread synchronization is needed to correctly compute the maximum scores in parallel. Synchronization can be achieved by atomic operations or by parallel reductions using *__syncthreads()* as explained in Section 3. The most viable synchronization method will of course vary depending on the mapping we choose. In practice, only atomic operations are an option in thread-based mapping, since we would otherwise need as many executions of parallel reductions, as the number of different parent symbols in each thread block. In block-based mapping, on the other hand, both parallel reductions and atomic operations can be applied.

4.3.1 Atomic Operations

In thread-based mapping, to correctly update the score, each thread needs to call the atomic API max operation with a pointer to the desired write location. However, this operation can be very slow (as we will confirm in Section 5), so that we instead perform a first reduction by calling the API with a pointer to the shared variable (as shown in line 14 of Figure 6), and then perform a second reduction with a pointer to the *scores* array (as shown in line 17 of Figure 6). 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.

In block-based mapping we can also use atomic operations on shared memory. However, in this mapping, 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 (as shown in line 15 of Figure 7). All the reductions are performed on this single shared variable. Compared to thread-based mapping, block-based mapping requires a fraction of the shared memory, and costly atomic operations on global memory are performed only once (as shown in line 18 of Figure 7).

4.3.2 Parallel Reductions

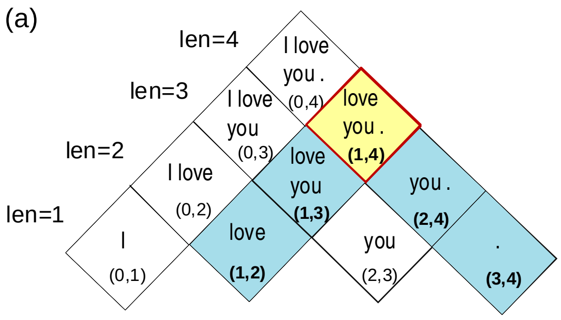
Parallel reduction is another option for updating scores in block-based mapping. Each thread in the thread block stores its computed score in an array declared as a shared variable and performs parallel reduction with a binary-tree order as shown in Figure 9 (from leaves to the root). When there are N threads in a thread block, the maximum score in the thread block is obtained after $\log 2N$ steps and stored in the first element in the array. Note that when 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. This approach can potentially be faster than the inherently serial atomic operations, but it comes with the cost of using more shared memory (proportional to the number of participating threads). This synchronization method is in practice only applicable to block-based mapping and cannot be applied to thread-based mapping since it assumes that all threads in a thread block perform reductions for the same parent symbol.

4.4 Reducing Global Memory Accesses

By now it should be clear that increasing CGMA and reducing global memory access is important for high GPU performance. We can approximately calculate the CGMA ratio of our kernel by counting global memory accesses and arithmetic operations per thread. There are three global memory accesses for each binary rule: the left child symbol ID, right child symbol ID, and the rule score itself. Moreover, there are two global memory accesses for referencing the scores with left child ID and right child ID in the split-point loop in line 7 of Figure 4. The loop in line 7 iterates up to the *length* of the span. The number of global memory accesses is thus $2 \cdot length + 3$. On the other hand, there are only two additions in the kernel, resulting in a very low CGMA. To improve performance, we need to increase the CGMA ratio by better utilizing the shared memory. Since shared memory is rather limited, and global memory accesses to the *scores* array in line 9 and 10 of Figure 4 spread over a wide range of memory locations, it is impossible to simply transform those global memory accesses into shared memory accesses. Instead, we need to modify the access pattern of the kernel code to meet this constraint.

If we look into the access pattern (ignoring the

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

Figure 10: (a) Chart example illustrating the access patterns of the *scores* array: the shaded cells are the locations that the threads with *start* = 1, *end* = 4 accesses. (b) Better memory access patterns with the new arrays

symbol dimension of the *scores* array), we can see that the accesses actually occur only in restricted locations that can be easily enumerated. For example, the accesses are restricted within the shaded cells in Figure 10(a) when the current cell is (1, 4). We can thus introduce two arrays (*sh_scores_L* and *sh_scores_R*) that keep track of the scores of the children in the shaded cells. These two arrays can easily fit into shared memory because there are only about 1000 unique symbols in our grammar and the sentence lengths are bounded, whereas the *scores* array can only reside in global memory. Figure 10(b) shows how the new arrays are accessed. For each unique left/right child symbol, we need to load the score from *scores* to *sh_scores_L* and *sh_scores_R* once through a global memory access. Thus, the number of global memory access will be reduced when multiple rules share the same child symbols.

Another way to reduce global memory accesses is to use texture memory. Recall that texture memory can be used for caching, but needs to be initiated from the CPU side and costs overhead. Moving the *scores* array to texture memory seems promising since it is frequently read to obtain the scores of children symbols. How-

CPU type	Core i7 2.80 GHz	
System memory	2GB	
GPU type	GTX285	GTX480
	648 MHz	1400 MHz
GPU memory	1GB	1.5GB
#SM	30	15
#SP/SM	8	32
Shared memory/SM	16KB	up to 64KB
L1 cache/SM	N/A	up to 64KB

Table 1: Experimental platforms specifications.

ever, as the array is updated at every iteration of binary relaxation, we need to update it also in texture memory by calling a binding API (between line 4 and 5 in Figure 3). While it can be costly to bind such a large array every iteration, we can reduce this cost by transforming its layout from *scores*[*start*][*end*][*symbol*] to *scores*[*length*][*start*][*symbol*], where $length = end - start$. With the new layout, we only need to bind the array up to size $(length - 1)$ (rather than the entire array), significantly reducing the cost of binding it to texture memory.

5 Experimental Results

We implemented the various versions of the parallel CKY algorithm discussed in the previous sections in CUDA and measured the runtime on two different NVIDIA GPUs used in a quad-core desktop environment: GTX285 and GTX480. The detailed specifications of the experimental platforms are listed in Table 1.

The grammar we used is extracted from the publicly available parser of Petrov et al. (2006). It has 1,120 nonterminal symbols including 636 preterminal symbols. The grammar has 852,591 binary rules and 114,419 unary rules. We used the first 1000 sentences from Section 22 of the Wall Street Journal (WSJ) portion of the Penn Treebank (Marcus et al., 1993) as our benchmark set. We verified for each sentence that our parallel implementation obtains exactly the same parse tree and score as the sequential implementation. We compare the execution times of various versions of the parallel parser in CUDA, varying the mapping, synchronization methods and memory access patterns.

Figure 11 shows the speedup of the different parallel parsers on the two GPUs: *Thread* stands for the thread-based mapping and *Block* for the block-based one. The default parallelizes over spans, while *SS* stands for sequential spans, meaning that the computation on spans is executed sequentially. *GA* stands for global atomic synchronization, *SA* for shared atomic synchronization, and *PR* for the parallel reduction. *SH* stands for the transformed access pattern for the *scores* array in the shared memory. *tex:rule* stands for loading the rule information from texture memory and *tex:scores* for loading the *scores* array from texture memory. *tex:both* means both the *tex:rule* and *tex:scores* are applied.

The exhaustive sequential CKY parser was written in C and is reasonably optimized, taking 5.5 seconds per sentence (or 5,505 seconds for the 1000 benchmark sentences). This is comparable to the better implementations presented in Dunlop et al. (2011). As can be seen in Figure 11, the fastest configuration on the GTX285 is *Block+PR+SS+tex:scores*, which shows a 17.4× speedup against the sequential parser. On the GTX480, *Block+PR* is the fastest, showing a 25.8× speedup. Their runtimes were 0.32 seconds/sentence and 0.21 seconds/sentence, respectively. It is noteworthy that the fastest configuration differs for the two devices. We provide an explanation later in this section.

On both the GTX285 and the GTX480, *Thread+GA* shows the worst performance as global atomic synchronization is very costly. *Thread+GA* on the GTX285 is even about 8 times slower than the sequential CKY parser. Note that although it is still the slowest one, *Thread+GA* on the GTX480 actually shows a 4.5× speedup.

On the GTX285, *Thread+SA*, *Block+SA*, and *Block+PR* show 6.4×, 8.1×, and 10.1× speedups, respectively. Perhaps somewhat surprisingly, parallelizing over spans actually hurts performance. By not serializing the computations for spans, we can get speedups of 13% for *Thread+SA+SS* over *Thread+SA* and about 40% for *Block+SA+SS* and *Block+PR+SS* over their parallel spans versions. In thread-based mapping, the atomic operations on shared memory are the bottleneck, so that sequential processing of spans makes only a small difference. On the other hand, in *Block+SA* and *Block+PR* on the GTX285, the global memory bandwidth is the major limiting factor since the same rule is loaded from the global memory redundantly for each span when we parallelize over spans. Hence, executing spans sequentially removes the redundant global memory loads and substantially improves the performance.

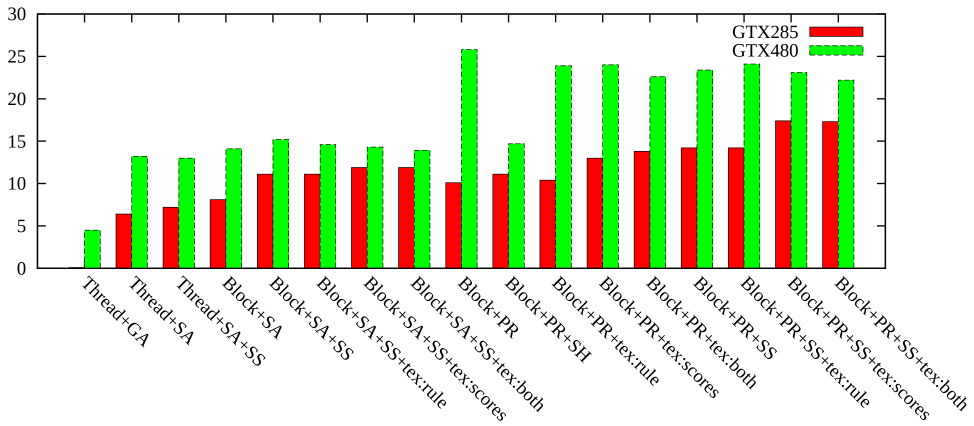


Figure 11: Speedups of various versions of parallel CKY parser that have different mappings, synchronization methods, and different memory access optimizations.

On the GTX285, the transformed access pattern for the *scores* array along with accesses to the shared memory (*Block+PR+SH*) improves the performance by about 10%, showing a 11.1× speedup. Placing the *scores* array in texture memory improves all implementations. The reduced binding cost due to the array reorganization results in additional gains of about 25% for *Block+PR+SS+tex:scores* and *Block+PR+tex:scores* against *Block+PR+SS* and *Block+PR* (for a total speedup of 17.4× and 13.0×, respectively). However, placing the rule information in texture memory improves the performance little as there are many more accesses to the *scores* array than to the rule information.

The GTX480 is the Fermi architecture (NVIDIA, 2009), with many features added to the GTX285. The number of cores doubled from 240 to 480, but the number of SMs was halved from 30 to 15. The biggest difference is the introduction of L1 cache as well as the shared memory per SM. For these reasons, all parallel implementations are faster on the GTX480 than on the GTX285. On the GTX480, parallelizing over spans (*SS*) does not improve the performance, but actually degrades it. This is because this GPU has L1 cache and a higher global memory bandwidth, so that reducing the parallelism actually limits the performance. Utilizing texture memory or shared memory for the scores array does not help either. This is because the GTX480 hardware already caches the scores array into the L1 cache.

Interestingly, the ranking of the various parallelization configurations in terms of speedup is architecture dependent: on the GTX285, the block-based mapping and sequential span processing are preferred, and the parallel reduction is preferred over shared-memory atomic operations. Using texture memory is also helpful on the GTX285. On the GTX480, block-based mapping is also preferred but sequential spans mapping is not. The parallel reduction is clearly better than shared-memory atomic operations, and there is no need for utilizing texture memory on the GTX480. It is important to understand how the different design choices affect the performance, since one different choices might be necessary for grammars with different numbers of symbols and rules.

6 Related Work

A substantial body of related work on parallelizing natural language parsers has accumulated over the last two decades (van Lohuizen, 1999; Giachin and Rullent, 1989; Pontelli et al., 1998; Manousopoulou et al., 1997). However, none of this work is directly comparable to ours, as GPUs provide much more fine-grained possibilities for parallelization. The parallel parsers in past work are implemented on multicore systems, where the limited parallelization possibilities provided by the systems restrict the speedups that can be achieved. For example, van Lohuizen (1999) reports a 1.8× speedup, while Manousopoulou et al. (1997) claims a 7-8× speedup. In contrast, our parallel parser is implemented on a manycore system with an abundant number of threads and processors to parallelize upon. We exploit the massive fine-grained parallelism inherent in natural language parsing and achieve a speedup of more than an order of magnitude.

Another difference is that previous work has often focused on parallelizing agenda-based parsers (van Lohuizen, 1999; Giachin and Rullent, 1989; Pontelli et al., 1998; Manousopoulou et al., 1997). Agenda-based parsers maintain a queue of prioritized intermediate results and iteratively refine and combine these until the whole sentence is processed. While the agenda-based approach is easy to implement and can be quite efficient, its potential for parallelization is limited because only a small number of intermediate results can be handled simultaneously. Chart-based parsing on the other hand allows us to expose and exploit the abundant parallelism of the dynamic program.

Bordim et al. (2002) present a CKY parser that is implemented on a field-programmable gate array (FPGA) and report a speedup of up to 750×. However, this hardware approach 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.

Ninomiya et al. (1997) propose a parallel CKY parser on a distributed-memory parallel machine consisting of 256 nodes, where each node contains a single processor. Using their parallel language, they parallelize over cells in the chart, assigning each chart cell to each node in the machine. With a grammar that has about 18,000 rules and 200 nonterminal symbols, they report a speedup of 4.5× compared to an optimized C++ sequential version. Since the parallel machine has a distributed-memory system, where the synchronization among the nodes is implemented with message passing, the synchronization overhead is significant, preventing them from parallelizing over rules and nonterminal symbols. As we saw, parallelizing only over chart cells (i.e., words or substrings in a sentence) limits the achievable speedups significantly. Moreover, they suffer from load imbalance that comes from the different number of nonteriminal symbols that each node needs to process in the assigned cell. In contrast, we parallelize over rules and nonterminal symbols, as well as cells, and address the load imbalance problem by introducing virtual symbols (see Figure 8).

It should be noted that there are a also number of orthogonal approaches for accelerating natural language parsers. Those approaches often rely on coarse approximations to the grammar of interest (Goodman, 1997; Charniak and Johnson, 2005; Petrov and Klein, 2007b). These coarse models are used to constrain and prune the search space of possible parse trees before applying the final model of interest. As such, these approaches can lead to great speed-ups, but introduce search errors. Our approach in contrast preserves optimality and could in principle be combined with such multi-pass approaches to yield additional speed improvements. There are also some optimality preserving approaches based on A*-search techniques (Klein and Manning, 2003; Pauls and Klein, 2009) or grammar refactoring (Dunlop et al., 2011) that aim to speed up CKY inference. We suspect that most of the ideas therein are orthogonal to our approach, and therefore leave their integration into our GPU-based parser for future work.

7 Conclusion

In this paper, we explored the design space for parallelizing the CKY algorithm for parsing, which is widely-used in constituency based natural language parsers. We compared various implementations on two recent NVIDIA GPUs. The fastest parsers on each GPU are different implementations, since the GTX480 supports L1 cache while the GTX285 does not, among other different architectural features. Compared to an optimized sequential C implementation our parallel implementation is 26 times faster on the GTX480 and 17 times faster on the GTX285. All our parallel implementations are faster on the GTX480 than on the GTX285, showing that performance improves with the addition of more Streaming Processors.

Acknowledgements

This work was supported in part by the 2010 Research Fund of the University of Seoul. This work was also supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional supports comes from UC Berkeley Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

We also thank Alexander (“Sasha”) Rush for valuable comments that lead to improvements of this paper.

References

K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. 2006. The landscape of parallel computing research: A view from berkeley. Technical report, Electrical Engineering and Computing Sciences, University of California at Berkeley.

J. Bordim, Y. Ito, and K. Nakano. 2002. Accelerating the CKY parsing using FPGAs. In *ICHPC '02*.

X. Carreras, M. Collins, and T. Koo. 2008. TAG, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *CoNLL '08*.

E. Charniak and M. Johnson. 2005. Coarse-to-Fine N-Best Parsing and MaxEnt Discriminative Reranking. In *ACL '05*.

E. Charniak. 2000. A maximum–entropy–inspired parser. In *NAACL '00*.

J. Cocke and J. T. Schwartz. 1970. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

M. Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA.

A. Dunlop, N. Bodenstab, and B. Roark. 2011. Efficient matrix-encoded grammars and low latency parallelization strategies for CYK. In *IWPT '11*.

J. Finkel, A. Kleeman, and C. D. Manning. 2008. Efficient, feature-based, conditional random field parsing. In *ACL/HLT '08*.

E. P. Giachin and C. Rullent. 1989. A parallel parser for spoken natural language. In *IJCAL '89*.

J. Goodman. 1997. Global thresholding and multiple-pass parsing. In *EMNLP '97*.

L. Huang and D. Chiang. 2005. Better k-best parsing. In *IWPT '05*.

T. Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab.

David B. Kirk and Wen-mei W. Hwu. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

D. Klein and C. Manning. 2003. A* parsing: Fast exact viterbi parse selection. In *NAACL '03*.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55.

A. G. Manousopoulou, G. Manis, P. Tsanakas, and G. Papakonstantinou. 1997. Automatic generation of portable parallel natural language parsers. In *Proceedings of the 9th conference on Tools with Artificial Intelligence*.

M. Marcus, B. Santorini, and M. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. In *Computational Linguistics*.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. 2008. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53.

T. Ninomiya, K. Torisawa, K. Taura, and J. Tsujii. 1997. A parallel cky parsing algorithm on large–scale distributed–memory parallel machines. In *PACLING '97*.

NVIDIA. 2009. Fermi: NVIDIA's next generation CUDA compute architecture. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

A. Pauls and D. Klein. 2009. Hierarchical search for parsing. In *NAACL-HLT '09*.

S. Petrov and D. Klein. 2007a. Discriminative log-linear grammars with latent variables. In *NIPS '07*.

S. Petrov and D. Klein. 2007b. Improved inference for unlexicalized parsing. In *NAACL '07*.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *ACL '06*.

E. Pontelli, G. Gupta, J. Wiebe, and D. Farwell. 1998. Natural language processing: A case study. In *AAAI '98*.

M. P. van Lohuizen. 1999. Parallel processing of natural language parsers. In *ParCo '99*, pages 17–20.

D.H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10.