Efficient Parallelization of Natural Language Applications using GPUs

Chao-Yue Lai

May 1, 2012

Acknowledgement

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

Efficient Parallelization of Natural Language Applications using GPUs

by Chao-Yue Lai

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kurt W. Keutzer
Research Advisor

(Sign and date)

Professor Marti A. Hearst
Second Reader

(Sign and date)

Abstract

As we enter the era of mobile computing, high-quality and efficient natural language applications become more and more important, which greatly facilitate intelligent human-computer interaction. Unfortunately, most high-quality natural language applications employ large statistical models, which render 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 natural language processing algorithms. In this report, we examine the possibility of parallelizing two major natural language applications, natural language parsing and machine translation on GPUs. In natural language parsing, we explore the design space of parallelizing the dynamic programming computations carried out by the CKY parsing algorithm. We use the Compute Unified Device Architecture (CUDA) programming model to re-implement 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 against an optimized sequential C implementation. In machine translation, we focus on parallelizing the CKY-based machine translation decoding algorithm using a phrase-based translation model and a trigram language model. Various optimization approaches exposing the inherent massive parallelism and reducing memory accesses have been investigated. Experimental results show that our best parallel implementation runs twice as fast as the optimized sequential implementation, without loss of accuracy. A runtime analysis shows that this suboptimal performance is caused by the memory-intensive nature and excessive amount of irregular memory accesses inherent in CKY-based machine translation decoding.

1. Introduction

Natural language applications are more and more important and prevalent nowadays. As the number of mobile devices surges, the demand for keyboard-less human-computer interaction, with natural language understanding as one of its core algorithms, also increases considerably. Siri, a feature of the Apple iPhone 4S, acts as a personal assistant to which users can speak commands ("Apple - iPhone 4S", 2011). Its natural language understanding algorithms parse commands and perform them accordingly. The emergence and huge success of Siri signifies the growing importance of natural language processing (NLP). Another prime example is IBM Watson, the machine built for answering questions in the Jeopardy! challenge ("IBM - What is Watson?", 2011). Watson defeated human champions (Jackson, 2011) in the Jeopardy! Show with its underlying DeepQA technologies (Ferrucci et al., 2010), where QA stands for Question Answering. The fact that IBM chose Jeopardy! as its grand challenge after building Deep Blue for chess, along with Watson's success, demonstrates the significance and maturity of NLP.

However, NLP algorithms are among the most computationally-challenging and thus time-consuming algorithms in computer science. In order to handle natural languages, large statistical grammar models and lexicons have to be built and consulted throughout NLP algorithms. The quality of NLP algorithms is proportional to the sizes of these statistical models, where large ones can consist of billions of words and grammar rules. Therefore, there is always a tradeoff between quality and running time in NLP applications, and for real-time applications, sacrifices in solution quality are often inevitable.

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 NLP applications, without sacrificing their quality. The new manycore machines, especially graphical processing units(GPUs), have been demonstrated to accelerate applications like computer vision (Catanzaro et al., 2009) and speech recognition (Chong et al., 2008), to more than an order of magnitude. It is also interesting to see if the same performance gain can be translated into NLP applications, whose computational patterns are vastly different from aforementioned application areas.

The two particular NLP applications we investigate in this report are natural language parsing and machine translation. Natural language parsing is the task of analyzing grammatical structures of an input sentence and predicting its most likely parse tree (see Figure 2), while in machine translation, input sentences from a foreign language is translated to a target language with the highest probability according to statistical language models and translation models. We identified the inherent concurrency in both applications and parallelized them on GPUs. We examined different design choices offered by the GPU architecture. In natural language parsing, the optimal implementation in GPU achieved a 26-fold speedup comparing to the original CPU implementation, without sacrificing accuracy, while in machine translation, a two-fold speedup is accomplished.

This report is structured as follows. In section 2, we present an overview of the general architecture of GPUs and efficient synchronization schemes provided by the Compute Unified Device Architecture (CUDA (Nickolls et al., 2008)) programming model. Section 3 and Section 4 discuss the details of parallelizing natural language parsing and machine translation, respectively. Both sections describe the respective NLP application, and its parallelization strategies, and then the experimental results, followed by related works. Finally, section 5 concludes the report.

2. 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 1: A group of SPs makes up a *streaming multiprocessor* (SM). A number of SMs forms 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.

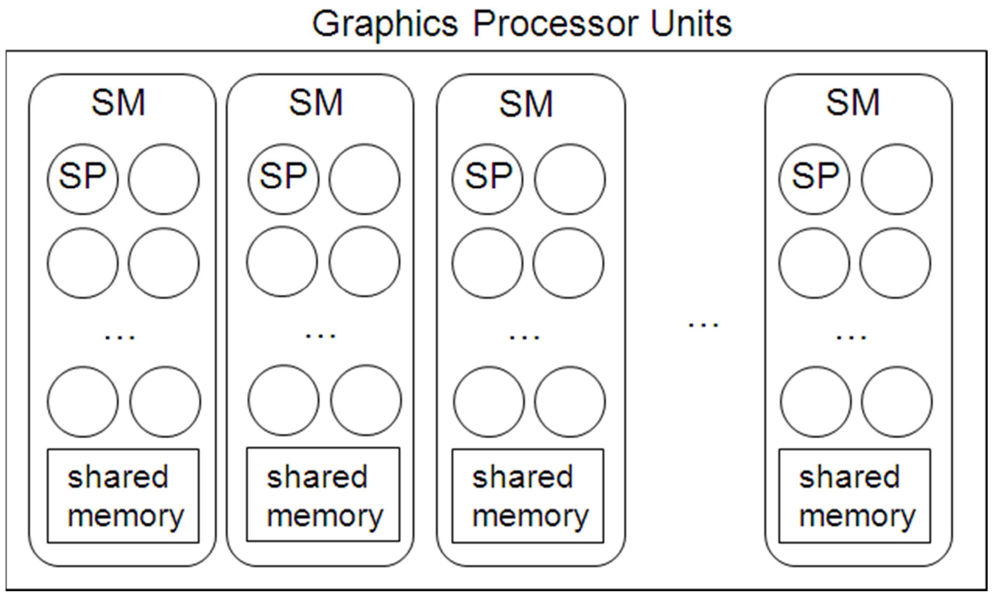


Figure 1: *Hierarchical structure of GPUs.*

2.1. 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. 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.

2.2. 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 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.

2.3. 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 4). 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.

2.4. 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 global barrier synchronization, one can use separate kernels when global barrier synchronization is needed.

3. Parallelization of Natural Language Parsing

Syntactic parsing of natural language is the task of analyzing the grammatical structure of sentences and predicting their most likely parse trees (see Figure 2). 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 non-terminal symbols and millions of context-free rules, while most sentences have on average only about n = 20 words. In Section 3.1, Context-free grammars and sequential CKY parsing algorithms are examined in more detail.

In Section 3.2, we present a general approach for parallelizing the CKY algorithm that can handle arbitrary CFGs. No assumptions is made about the size of the grammar and the efficacy of our approach is demonstrated 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 discuss how the

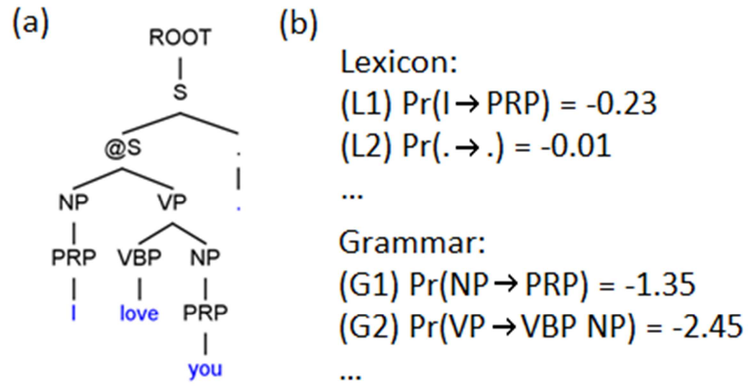


Figure 2: *An example of a parse tree for the sentence "I love you ."*

hundreds of cores available on a GPU can enable a fine-grained parallel execution of the CKY algorithm. We then explore the design space with different thread mappings onto the GPU and discuss how the various synchronization methods might be applied in this context. 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 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 3.3. 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 3.4). 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.

3.1. Natural Language 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.

3.1.1. 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). The grammars in our experiments have on the order of thousands of non-terminals and millions of productions.

Figure 2(a) shows a constituency parse tree. Leaf nodes in the parse tree, also called terminal nodes, correspond to words in the language. Pre-terminals correspond to part-of-speech tags, while the other non-terminals correspond to phrasal categories. For ease of exposition, we will say that terminal productions are part of a lexicon. For example, (L1) in Figure 2(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*.

3.1.2. 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 3). After initializing the pre-terminal 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 (pseudo-code is given in Figure 4). To reconstruct the highest scoring parse tree we perform a top-down search. We found this to be more efficient than keeping backpointers.

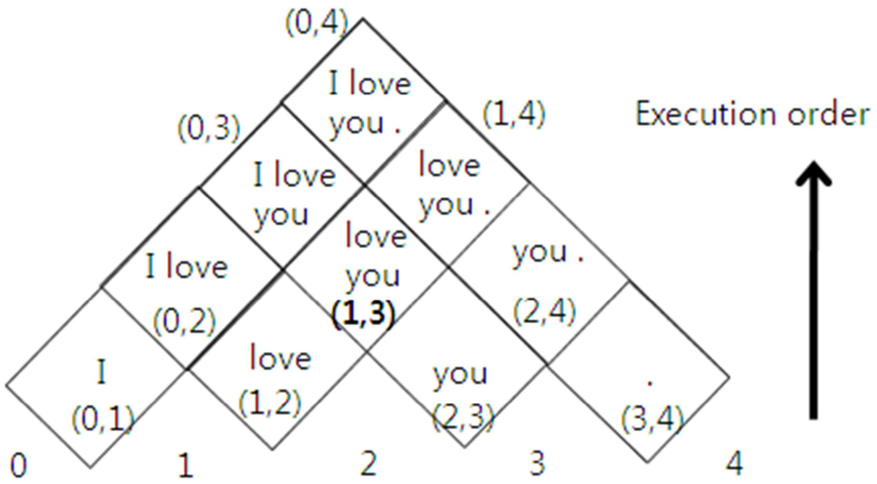


Figure 3: *The chart that visualizes the bottom-up process of CKY parsing for the sentence "I love you ."*

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

```
nWords = readSentence(sen);
scores[][][] = initScores(nWords);
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 4: *Pseudo-code for CKY parsing*

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 extraction 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.

3.2. Parallel Natural Language Parsing on GPUs

The dynamic programming loops of the CKY algorithm provide various types of parallelism. While the loop in Figure 4 cannot be parallelized due to dependencies between iterations, all four loops in Figure 5 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.

Algorithm: **`binaryRelax`**`(scores, nWords, length, gr)`
Input: `scores` /* the 3-dimensional scores array */
`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 5: *Binary relaxations in CKY parsing.*

3.2.1. 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 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, in doing so, it not only fails to provide enough parallelism to fully utilize the massive number of threads in GPUs, but it can also suffer from load imbalance since 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 greatly. It is therefore advantageous to map rules rather than symbols to threads.

3.2.1.1. Thread-Based Mapping

If we map rules to threads, the nested loops in line 3 and line 5 of Figure 5 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 18 of Figure 6), which can be costly.

3.2.1.2. 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 relatively fewer rules, it can proceed to the next available thread block, which corresponds to a different symbol.

Algorithm: **threadBasedRuleBR**(*scores*, *nWords*, *length*, *gr*)
Input: *scores* /* the 3-dimensional scores array */
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__ float 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
16 // global maximum reduction
17 foreach symbol in gr in parallel
18   atomicMax(&scores[start][end][symbol],
                                sh_max[symbol]);
```

Figure 6: *Thread-based parallel CKY parsing.*

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 pseudo-code 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 pre-terminal symbols (i.e. part-of-speech tags), can only cover spans of length 1 (i.e. single words). In block-based mapping,

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

for start = 0 to nWords - length in parallel
  end = start + length;
  foreach symbol in gr in parallel
    __shared__ float sh_max = FLOAT_MIN;
    foreach rule r per symbol in parallel
      // 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;
        // local maximum reduction
        if score > local_max
          local_max = score;
      atomicMax(&sh_max, local_max);

// global maximum reduction
foreach symbol in gr in parallel
  atomicMax(&scores[start][end][symbol],
                                             sh_max);
```

Figure 7: *Block-based parallel CKY parsing*

only one thread needs to check and determine if a symbol is a pre-terminal and can be skipped. In contrast, in thread-based mapping, every thread in the 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 the 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.

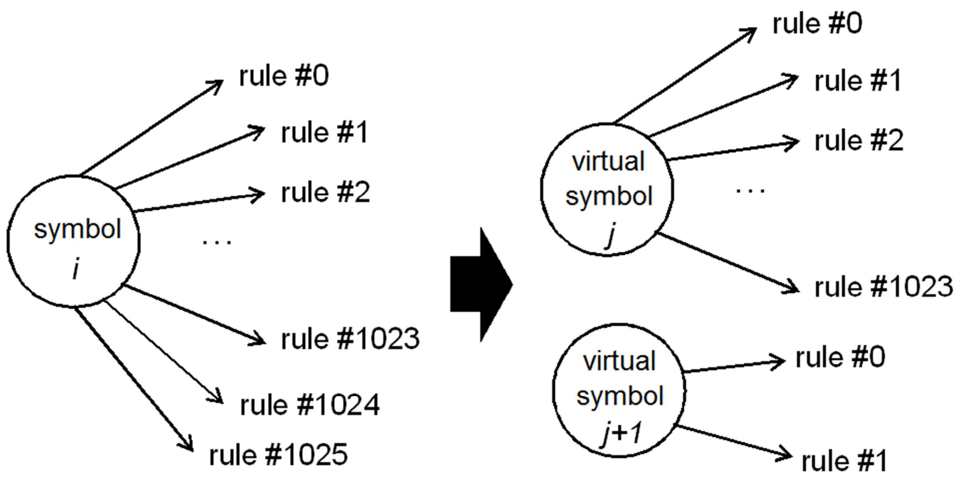


Figure 8: *Creating virtual symbols whenever a symbol has too many rules.*

3.2.2. 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 3), 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.

3.2.3. 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 2. 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.

3.2.3.1. 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 18 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 19 of Figure 7).

3.2.3.2. 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

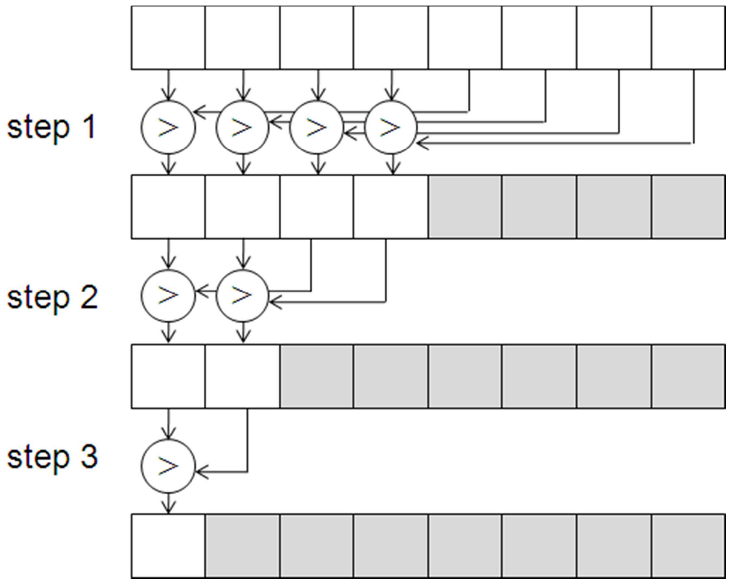


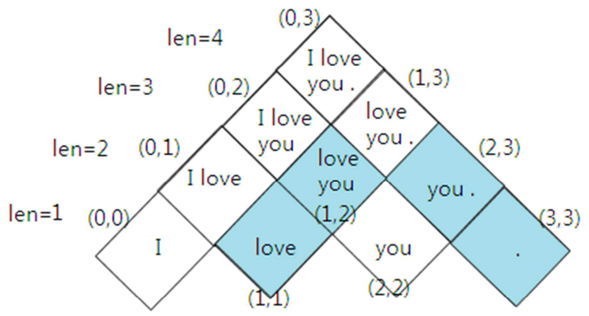
Figure 9: *Parallel reduction on shared memory between threads in the same thread block with 8 threads.*

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.

3.2.4. Reduce 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.

(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*

If we look into the access pattern (ignoring the *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. However, 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 4). 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.

3.3. 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.

CPU type	Core i7 2.8GHz	
System Memory	2GB	
GPU Type	GTX285 648MHz	GTX480 700MHz
GPU Global 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: *Specifications of Experimental Platforms for CKY parsing*

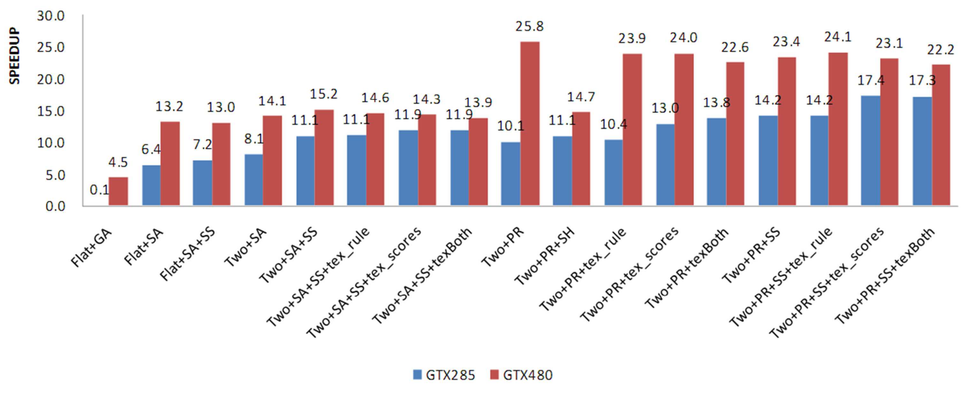


Figure 1: *Speedups of various versions of parallel CKY parser that have different mappings, synchronization methods, and different memory access optimizations.*

The grammar we used is extracted from the publicly available parser of Petrov et al. (2006). It has 1,120 non-terminal symbols including 636 pre-terminal 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.

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 an 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 (“Fermi: NVIDIA’s Next”, 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 choice might be necessary for grammars with different numbers of symbols and rules.

3.4. 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 non-terminal 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 non-terminal 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 non-terminal symbols that each node needs to process in the assigned cell. In contrast, we parallelize over rules and non-terminal 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 also a 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.

4. Parallelization of Machine Translation

Machine translation (MT) is one of the most computationally challenging problems in the field of natural language processing. Statistical machine translation (SMT) uses the knowledge from trained language and translation models to automatically translate sentences between languages. Translating each sentence requires large amounts of unpredictable data accesses to the language model and the translation model, making this a very memory-intensive application, whose data access pattern is determined at runtime. Translation process is slow and accuracy greatly depends on the size of the language and translation models used. Efficient parallelization of machine translation will open up new possibilities in many fields of general-purpose computing including real-time human computer interaction, multimedia systems and surveillance systems.

As a complicated problem machine translation is, SMT algorithms vary in terms of granularity of translation units. In our work, we adopted the phrased-based translation paradigm as developed in Koehn et al. (2003). SMT also has many variations in strategies to form larger translations from smaller ones. We adopted a CKY-based chart decoding strategy, which is similar to the CKY parsing algorithm described in the last section.

More detailed descriptions of the chosen SMT algorithm will come in Section 4.1. In Section 4.2, we describe various strategies to parallelize this inherently-memory-intensive algorithm. The experimental results are given in Section 4.3, where the best parallel implementation is twice faster than the sequential implementation. We will give profiling results to show that the suboptimal performance is mainly caused by overheads of memory accesses and divergent control branches. Finally, with sufficient knowledge about different variations of SMT algorithms, related works are introduced in Section 4.4.

4.1. Statistical Machine Translation

In this section we give an overview of the sequential SMT algorithm. We first give an overview of an SMT system, and then we introduce individually the two key statistical models in SMT: language model and translation model. The CKY-based decoding algorithm upon which we parallelize is explicated afterwards.

4.1.1. Overview

In statistical machine translation, given a foreign sentence **f**, we want to find an English sentence **e**, such that its probability is maximized:

$$\mathrm{argmax}_{\mathrm{e}}\ p(\mathbf{e}|\mathbf{f})$$

With the help of the Bayes rule, the translation probability can be reformulated as:

$$\mathrm{argmax}_{\mathbf{e}}\ p(\mathbf{e}|\mathbf{f}) = \mathrm{argmax}_{\mathbf{e}}\ p(\mathbf{f}|\mathbf{e})p(\mathbf{e})$$

This allows for a language model $p(\mathbf{e})$ and a separate translation model $p(\mathbf{f} \mid \mathbf{e})$.

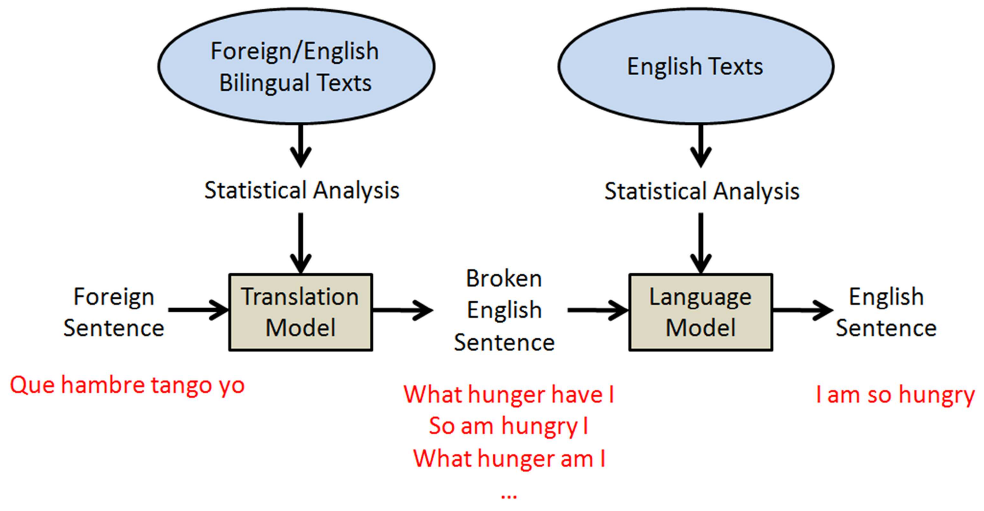


Figure 12: *An overview of an SMT system, with a Spanish-to-English example.*

Figure 12 gives an overview of an SMT system. Given a foreign sentence, the system first uses the translation model to generate a collection of English segments, and then the language model is consulted to reassemble the fragments into an English sentence with the highest probability. The whole translation process utilizing both models is called *decoding*, and how the two models are utilized is called the *decoding algorithm*. These terms are coined to distinguish from the statistical analyses used to generate the two models, on which most MT research focuses.

4.1.2. Language Model

A language model (LM) assigns probabilities to word sequences in order to account for their relative frequencies with one another in a natural language. For example, the English word "he" occurs much more frequently than "statistical". We will thus assign a much higher score to "he" than to "statistical".

A language model can also be called an *n*-gram model, where *n* is the maximum number of words that this LM assigns probability to. An *n*-gram language model is essentially an (n – 1)-order Markov model. In this work, we employed a *trigram* model, where we assign probability to a word given its (at most) two preceding words, as the nature of 2nd-order Markov model. A trigram model suffices since there is no significant improvement of accuracy from a trigram to a four-gram model.

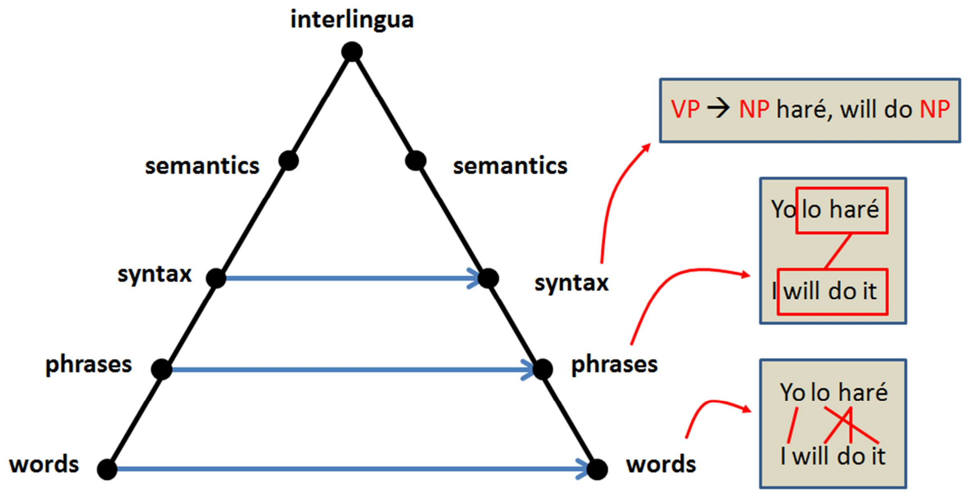


Figure 13: *levels of transfer: different levels of granularity on which translation models are based. More linguistic understanding is required as the pyramid goes up. Current SMT algorithms are based on the bottom three levels, while the ultimate goal is to use interlingua to represent meanings independent of languages.*

4.1.3. Translation Model in Phrase-based Machine Translation

A translation model (TM) assigns probabilities to pairs of foreign-English expressions, which reflect the relative frequencies of co-occurrences of the pairs in a bilingual corpus. For example, the Spanish word "sombrero" means "hat" in English in most contexts. It is thus very likely that the Spanish-English pair (sombrero, hat) has a higher probability than, say, (sombrero, juice).

The forms of expressions a TM assigns probabilities to depend on the level of transfer of the TM, which is illustrated as the pyramid in Figure 13. In word-based MT paradigm, only frequencies of word pairs are evaluated, whereas in phrase-based MT paradigm, phrase pairs are the fundamental unit. In syntactic MT paradigm, part-of-speech tags are employed to allow more linguistic information to enter the model.

Most state-of-the-art translation-models are phrase-based (Koehn, 2004; Tillmann, 2003), but syntactic MT is also a promising area of research (Chiang, 2005; Yamada and Knight, 2002), on which translation of language pairs with significantly different word orderings, such as Chinese and English, performs better. In this work, we adopted phrase-based translation models. In particular, the length of phrases is limited to three since there is no significant performance gain including longer phrase pairs (Koehn et al., 2003).

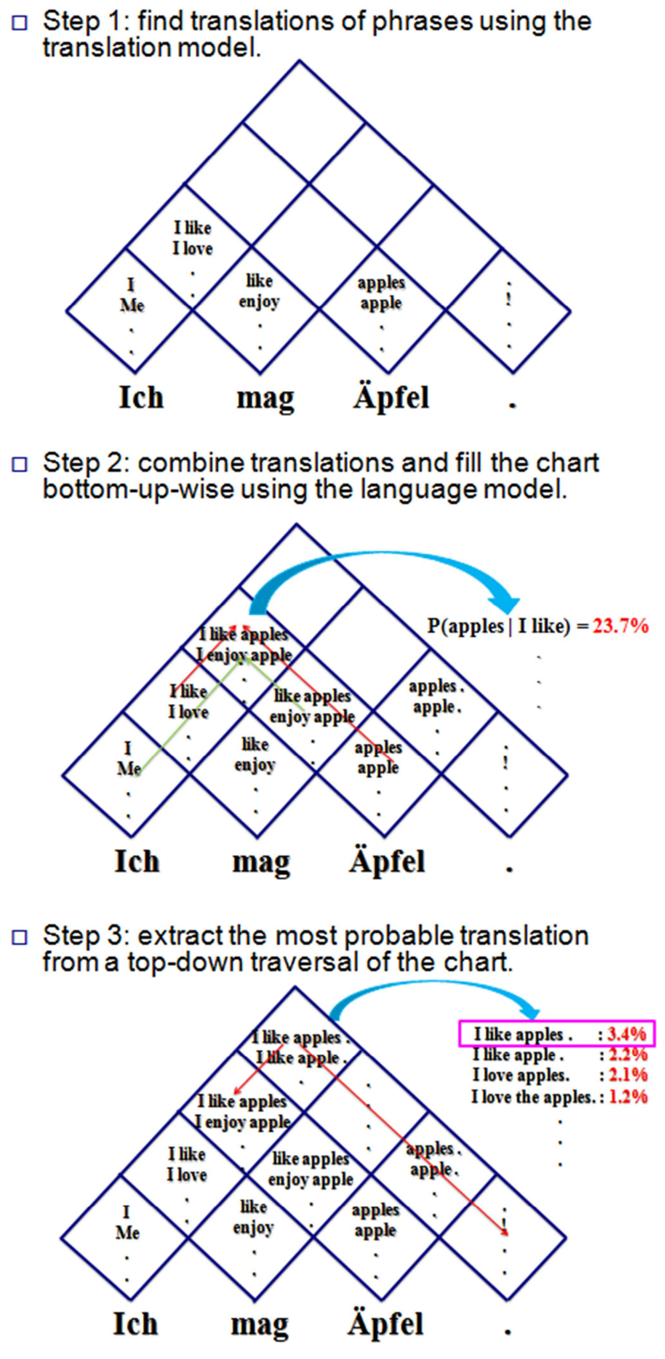

Step 1: find translations of phrases using the translation model.
I Me
I like I love
like enjoy
apples apple
!
Ich mag Äpfel .
Step 2: combine translations and fill the chart bottom-up-wise using the language model.
P(apples | I like) = 23.7%
I like apples I enjoy apple
like apples enjoy apple
apples . apple .
Ich mag Äpfel .
Step 3: extract the most probable translation from a top-down traversal of the chart.
I like apples . I like apple .
I like apples . : 3.4%
I like apple . : 2.2%
I love apples. : 2.1%
I love the apples. : 1.2%
Ich mag Äpfel .

Figure 14: *The three steps of a CKY-based decoding algorithm, which is very similar to chart parsing.*

```
Algorithm: translate(sen, lm, tm)
Input: sen /* the input sentence */
       lm /* the language model */
       gm /* the translation model */
Output: trans /* the most probable English translation */

1  nWords = readSentence(sen);
2  cells[][] = initCells(nWords);
3  translatePhrases(cells, sen, nWords, tm);
4  for length = 2 to nWords
5    combineTrans(cells, nWords, length, lm);
6  translation = backtrackBestTrans(cells);
7  return translation;
```

Figure 15: *pseudo-code for CKY-based MT decoding algorithm.*

4.1.4. CKY-based Decoding Algorithm

We adopt a CKY-based decoding algorithm, which is very similar to the CKY chart parsing algorithm mentioned in Section 3.1.2. Figure 14 shows the three major steps of this algorithm, while Figure 15 provides its pseudo-code. First, we initialize the two-dimensional *cells* data structure in line 2, which represents the chart pyramid in Figure 14. We then apply the translation model to find translations of words and phrases up to length three, to fill up the bottom three levels of the chart (step 1 in Figure 14, line 3 in Figure 15). As follows, we combine translations of smaller lengths to larger lengths in a bottom-up dynamic-programming fashion (step 2, lines 4-5). Finally, the best translation *translation* is extracted from a top-down backtracking in *cells* (step 3, line 6).

The CKY-based decoding algorithm in Figure 15 and the CKY parsing algorithm in Figure 4 are very similar. Basically, LM and TM in CKY decoding correspond to lexicon and grammar in CKY parsing, respectively. One major difference is that there is no unary relaxation in CKY decoding. The *combineTrans* function corresponds to binary relaxation, since it is always combining two lower cells. Another important difference is the dimensionality of the main data structure, *scores* in parsing and *cells* in MT decoding. The *scores*[*start*][*end*][*symbol*] array is three-dimensional, where the third dimension corresponds to *symbol*, since one chart cell can only be mapped to one *symbol* and the number of *symbols* is limited to a few hundreds. Whereas in MT decoding, *cells*[*start*][*end*] is two-dimensional.

In each cell, a list of possible translations for words in the range of (*start*, *end*) is stored. Unlike in parsing, one cell can potentially correspond to infinite number of translations, which are combinations of English words in the LM and TM. We thus cannot index the translations like we do for *symbols* in parsing. We also have to limit the number of translations per *cell* and prune unlikely ones.

Just like binary relaxation is the bottleneck in CKY parsing, combining translations is the bottleneck in CKY-based MT decoding, as its details are discussed in the next subsection.

```
Algorithm: combineTrans(cells, nWords, length, lm)
Input: cells /* the 2-dimensional cells array */
       nWords /* the number of total words */
       length /* the current span */
       lm /* the language model */
Output: None

for start = 0 to nWords - length
  end = start + length;
  cur_cell = cells[start][end];
  for split = start + 1 to end - 1
    l_list = cells[start][split];
    r_list = cells[split][end];
    foreach l_trans in l_list
      foreach r_trans in r_list
        // calculate LM score
        lm_score = getLMscore(l_trans,
                                     r_trans, lm);
        score = l_trans.score + r_trans.score
                                     + lm_score;
        new_trans = createTrans(l_trans,
                                 r_trans, score);
        addTrans(cur_cell, new_trans);

  // pruning unlikely translations
  max = getMaxScore(cur_cell);
  threshold = max - BEAM_WIDTH;
  pruneCell(cur_cell, threshold);
```

Figure 16: *combineTrans in CKY-based MT decoding.*

4.1.4.1. Combining Translations

Figure 16 shows the pseudo-code for *combineTrans* in line 5 of Figure 15. The outer-most loop iterates through all cells in one level of the chart given a specific *start* position. Given a *cell* (line 3), we iterate through different *split* positions, and combine each translation in the left cell (as *l_list*) with each translation in the right cell (as *r_list*), by calculating the LM score according to words in the boundaries of the translations (line 10) and adding the scores up (line 11). We collect all new combined translations with all *split* positions and put them in the target *cur_cell*.

We also prune unlikely translations with a beam search strategy (lines 16-18). This is also done in other major MT decoder systems for efficiency (Koehn et al., 2003). The beam width can be chosen to reflect the tradeoff between speed and accuracy.

4.1.4.2. Simplifying Translations into Contexts

A key observation to the efficient implementation of CKY-based decoding algorithm is that not all words in a translation are necessary to calculate LM scores. As shown in Figure 17, with a trigram LM, only the first two and the last two words of translations are required. Therefore, we only need to keep (at most) four words in a translation, defined as the *context*. Simplifying translations into contexts largely reduce memory usage and creates a more regular data structure, both of which are beneficial in GPU programming.

With the use of contexts, there will be much more reduplicated translations inside a cell. We can eliminate duplicates of contexts with lower scores to accelerate the algorithm without loss of accuracy.

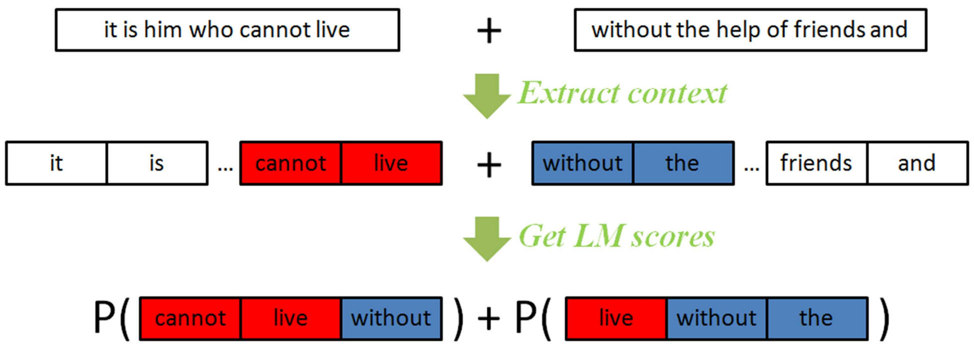

it is him who cannot live
+
without the help of friends and
Extract context
it
is
...
cannot
live
+
without
the
...
friends
and
Get LM scores
P(cannot live without) + P(live without the)

Figure 17: *We only need the first two and the last two words of translations (defined as the contexts) to calculate the LM scores in a trigram model.*

```
Algorithm: combineTransInParallel(cells, nWords,
length, lm)
Input: cells /* the 2-dimensional cells array */
       nWords /* the number of total words */
       length /* the current span */
       lm /* the language model */
Output: None

for start = 0 to nWords - length in parallel
  end = start + length;
  cur_cell = cells[start][end];
  for split = start + 1 to end - 1 in parallel
    l_list = cells[start][split];
    r_list = cells[split][end];

    __shared__ sh_r_list;
    copyToShared(r_list, sh_r_list);
    foreach l_trans in l_list in parallel
      foreach sh_r_trans in sh_r_list
        // calculate LM score
        lm_score = getLMscore(l_trans,
                                sh_r_trans, lm);
        score = l_trans.score +
                  sh_r_trans.score + lm_score;
        new_trans = createTrans(l_trans,
                                sh_r_trans, score);
        atomicAddTrans(cur_cell, new_trans);

  uniquifyCellInParallel(cur_cell);

  // pruning unlikely translations
  max = getMaxScoreInParallel(cur_cell);
  threshold = max - BEAM_WIDTH;
  pruneCellInParallel(cur_cell, threshold);
```

Figure 18: *pseudo-code for combineTransInParallel.*

4.2. Parallel Machine Translation on GPUs

In this section, we will focus on various approaches and design choices to efficiently parallelize combining translations, *combineTrans*, where the majority of runtime of the whole algorithm is spent on.

4.2.1. Thread Mapping

Figure 18 shows the pseudo-code of the parallel version of *combineTrans*. As in the case of parallelizing parsers, all four loops in *combineTrans* are parallelizable. However, we do not parallelize upon the inner-most loop (line 11), because this loop is not completely independent of the previous loop (line 10). Given a split point (line 4), the two inner-most loops iterate through lists of left and right translations, respectively. For each pair of left and right translations, we will calculate the LM score of the combination and add this new translation to the current cell (lines 13-16). That is to say, if there are *l* and *r* translations in *l_list* and *r_list*, respectively, we will generate *l·r* new translations.

We can certainly assign each thread to perform one combination of left and right translations, by making the two inner-most loops executed in parallel. However, this might induce load imbalance among threads, since the overhead of *getLMscore* critically depends on its input words[1]. Moreover, the amount of work performed per thread will be too little to justify the overhead of parallelism and inter-thread synchronization.

Instead, we copy all the translations in *r_list* in the shared memory (lines 8-9) to share among threads in a thread block. We let one thread to handle the combinations of one left translation with all corresponding right translations. This significantly increases the amount of work per thread and is more load-balanced.

4.2.2. Span-Level Parallelism

Just like in parallelizing parsers, spans with the same length are independent of one another and thus can be processed in parallel (line 1 in Figure 18). Again, we utilize the multi-dimensional indexing of thread blocks in CUDA to achieve this task. We assign the *x*-dimension of the thread block to different start positions in a level (line 1) and the *y*-dimension to different split points given a specific start position (line 4). By doing so, a thread block handles the combination of

[1] For example, if the input three-word sequence does not exist in the trigram model, we have to back off to bigram probabilities and/or to adopt the probability for unknown words.

translations of left and right lists given a split point. Then, we copy all the right translations onto the shared memory and map a thread to a left translation (line 10). The rest of the process is explained previously. We will show in the experiments that exploiting this additional level of parallelism significantly improves the performance of the parallel MT decoder.

We have to pay extra attention when we add new translations to the target cell (line 16). Since every thread in a thread block updates the same target cell *cur_cell*, synchronization is necessary to avoid inconsistency. In the actual code, we use a shared variable *cur_location* to keep track of the end of the list of *cur_cell*. When we add translations, we first use *atomicAdd* to increment *cur_location*, and then we put the new translation at *cur_location*.

4.2.3. Uniquifying and Pruning Translations

At the end of combining translations, new translations that have the same contexts have to be trimmed (called "uniquifying") and translations that fall out of beam have to be pruned. Uniquifying and pruning translations are indispensible for the efficiency of the CKY-based MT decoding algorithm. These two procedures can also be performed in parallel (lines 18-23 in Figure 18), as explained as follows.

In uniquifying translations, we can compare the contexts of each pair of translations and eliminate the translation with a lower score/probability. If the number of translations is n, This pair-wise comparison is essentially of O(n^2) complexity, and not parallelizable. We thus do not adopt the pair-wise comparison method.

Instead, we sort the translations according to the words in the contexts and then remove consecutive translations with the same contexts. The complexity of this sort-and-remove method lowers to O($n$log n). Furthermore, both sorting and removing consecutive duplicates are parallelizable. In this work, we adopt the optimized implementations of both routines in the CUDA programming library Thrust (Hoberock and Bell, 2000).

Pruning can be divided into two major steps, maximization of scores (line 21) and eliminating translations with low scores (line 23). Both of them are amenable to parallelization. Maximization of scores can be either implemented with atomic operations or parallel reductions, with little influence on running time. Eliminating elements in a vector with a threshold is a well-defined parallel function, and we adopt the efficient Thrust function *reduce_by_key* to perform the task.

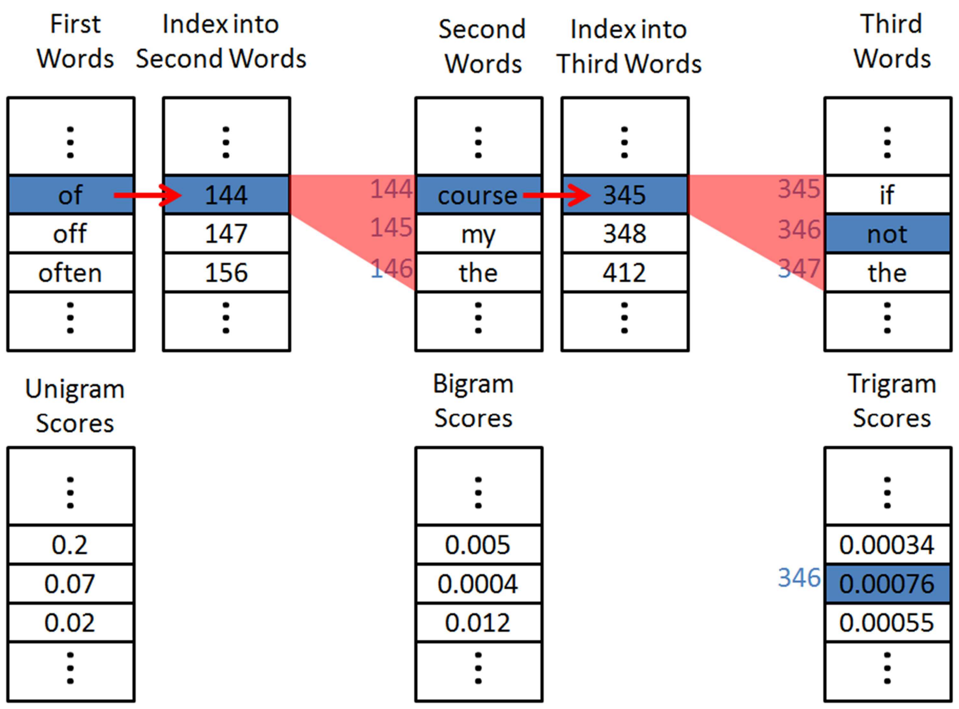


Figure 19: *An LM score table with compact indexing, with an example of looking up the trigram probability of "of course not." Notice that in the actual implementation the words are replaced with IDs for simplicity and efficiency.*

4.2.4. Implementations of LM Score Table

The fastest and simplest way to implement a trigram LM score table is to use a three-dimensional array, where indices are word IDs. However, given the large number of words in language models, a 3D array is not practical in CPUs, not to mention GPUs, whose memory is rather limited. Furthermore, the 3D array will be extremely sparse, since most word sequences are not common in natural languages. We need a more memory-efficient implementation of the LM score table.

The CPU implementation of the LM score table is the C++ standard library <*map*>, with the triplet of words as the key and the LM score as the value. However, this implementation is not viable on GPU, because <*map*> is not thread-safe, which may induce race conditions. Moreover, the binary-tree implementation of <*map*> will create too many global memory accesses on GPU for the lookups to be efficient.

We can condense the three-dimensional LM score table by recording all the word indices that can follow a unigram/bigram in a vector. Figure 19 is an example of such compact LM score table. We have three vectors for word IDs, *First_Words*, *Second_Words*, and *Third_Words*. The vector *Index_into_Second_Words* records the starting index in *Second_Words* where the single word can follow. For example, the index of "of" in *Index_into_Second_Words* is 144, and the next index is 147, which means there are 3 (147 - 144) bigrams starting with "of", and the second words are stored in *Second_Words* of indices 144 to 146.

Index_into_Third_Words and *Third_Words* are designed with the same methodology. For example, if we want to search for the trigram score of "of course not", we first check the index of "of" in *Index_into_Second_Words* and the index of its next word, and get the range 144 to 146. Then we search through *Second_Words* for the word ID of "course", and check its corresponding index in *Index_into_Third_Words* (and its next index), and get the range 345 to 347. Finally, we search through *Third_Words* for the word ID of "not", and access the corresponding score in *Trigram_Scores*.

We search through *Second_Words* and *Third_Words* with a binary search instead of a linear search, given that the word IDs pertaining to the same unigram/bigram are stored in ascending order. Binary searches largely reduce global memory accesses comparing to linear searches, which is crucial for GPU performance, especially when the bigram/trigram is not found in the language model.

Another possible implementation is a parallel hash map. Alcantara et al. (2009) demonstrates a construction method of hash maps on GPUs with a hybrid approach of sparse perfect hashing and cuckoo hashing. Comparing to the LM score table with compact indexing, the number of memory accesses does not vary considerably with the input words. Nevertheless, although sparse perfect hashing is used to construct the hash map, the load factor is 71% (Alcantara et al. 2009), with a 29% waste of space. This is unfavorable under a memory-limited GPU setting, comparing the compact-indexing LM score table.

4.3. Experimental Results

We have implemented the parallel CKY MT decoder in CUDA, with various design choices discussed in the last subsection. The runtimes and speedups are measured on one NVIDIA GPU, GTX480, in a quad-core desktop environment, against a serial implementation running on a Core i7 quad-core processor, as listed in Table 2. Notice that GTX285 is not tested since we arrived at the same conclusions comparing two GPUs as in the case of parallel CKY parsing.

CPU type	Core i7 2.8GHz quad-core
System Memory	2GB
GPU Type	GTX480 700MHz
GPU Global Memory	1.5GB
#SM	15
#SP/SM	32
Shared Memory/SM	up to 64KB
L1 Cache/SM	up to 64KB

Table 2: *Specifications of Experimental Platforms for MT decoding*

The phrase-based translation model and the trigram language model are trained using the Spanish-English parallel texts in the EuroParl corpus (Koehn 2005). The translation model is trained by Moses (Koehn et al., 2007), an open-source SMT toolkit for training translation models and decoding. The translation model consists of 5,041,515 Spanish-English rules, where only unigram, bigram and trigrams are considered on both sides. The language model is trained with SRILM (Stolcke, 2002), an open-source language modeling toolkit, where we limit the length of word sequences to trigrams. The resulting LM has 117,579 unigrams (and thus, words), 2,690,679 bigrams and 2,863,446 trigrams. All of the words appearing in the English side of the TM are captured in the LM, since they are trained on the same corpus.

We used the first 1000 pairs of sentences in the Spanish-English texts in the EuroParl corpus (Koehn 2005) as our benchmark set. Identicalness of the resultant translation between serial and parallel implementations cannot be guaranteed due to the inherent randomness of parallel execution. Nevertheless, we verified that translation quality does not degrade with parallelization, by asserting that there is no decrease of BLEU score (Papineni et al, 2002), a *de facto* MT evaluation standard.

We first tried with different design choices on span-level parallelism as well as uniquifying. The results are summarized in Table 3. As we can see, all results with parallel spans greatly outperform those with sequential spans. This is not surprising since much more parallelism available in the CKY decoding is exploited in parallel spans.

Span	Uniquify	Runtime (seconds)	Speedup
Sequential	No	2320.0	0.1
Sequential	Pair-wise	590.0	0.4
Sequential	Sorting	333.3	0.7
Parallel	No	468.1	0.5
Parallel	Pair-wise	147.9	1.6
Parallel	**Sorting**	**117.6**	**2.0**

Table 3: *the runtime and speedup comparisons among various design choices over 1000 sentence pairs. The sequential implementation runs in 235.7 seconds.*

Uniquifying can be implemented with pair-wise comparison, parallel sorting, or not realized at all. From Table 3 it is obvious that uniquifying is mandatory for practical reasons. Without uniquifying, the numbers of translations per cell increases considerably, which greatly slow down the decoding process. Parallel sorting is preferred to pair-wise comparisons, as Thrust provides an optimized implementation.

On the other hand, no significant difference in runtime is observed with different implementations of the LM score table, as shown in Table 4. It seemed that parallel hash maps should perform better, since the number of memory accesses remains constant with respect to different input words, unlike with compact indexing. However, compact indexing has fewer memory accesses when the input trigram is not in the model. As a result, compact indexing slightly outruns parallel hash maps, even when the latter is implemented in texture memory.

Compact Indexing	Parallel Hash Map	Runtime (seconds)	Speedup
Yes	**N/A**	**117.6**	**2.0**
No	Global Memory	130.2	1.8
No	Texture Memory	124.0	1.9

Table 4: *the runtime and speedup comparisons with different implementations of the LM score table over 1000 sentence pairs. The sequential implementation runs in 236 seconds*

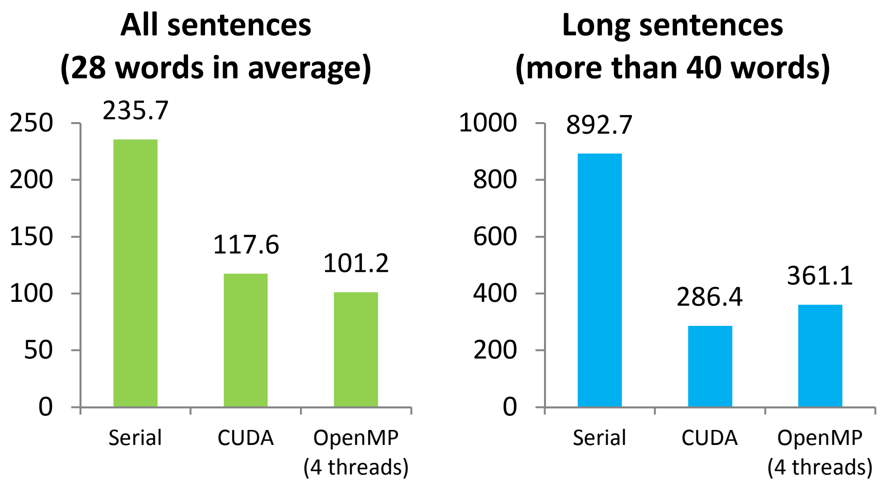


Figure 20: *the runtime comparisons (in seconds) of different parallelization schemes, CUDA versus OpenMP, on 1000 sentences with different lengths.*

The 2.0× speedup over serial implementation is clearly suboptimal. Figure 20 demonstrates this by comparing the best CUDA implementation with an OpenMP implementation (Dagun and Menon, 1998), which is tested on the quad-core Core i7 CPU shown in Table 2. With all sentences, the CUDA implementation runs slower than the OpenMP one, even when the latter is much easier to program and only uses four threads. However, we can see in the right graph of Figure 20 that CUDA performs much better on longer sentences in comparison to OpenMP, where CUDA achieves a 3.0× speedup. Translating longer sentences benefits more from span-level parallelism, since they have bigger charts with more independent tasks to saturate the GPU cores and threads.

The profiling results of our CKY-based MT decoder in Figure 21 point out the reason why only limited speedup can be achieved. *Add Scores* in Figure 21, or line 14 in Figure 18, is the only compute-intensive part of the entire *combineTransInParallel* method. As we can see, *Add Scores* only accounts for 5% of the total runtime, and all the rest parts are memory-intensive[2]. This low compute-to-memory ratio does not favor GPU computing.

[2] *Get LM Scores* is a series of table lookups, no matter whether compact indexing or hash maps are used. *Uniquify* and *Prune* are both heavy on comparison of elements and relocating them. None of them contain arithmetic instructions.

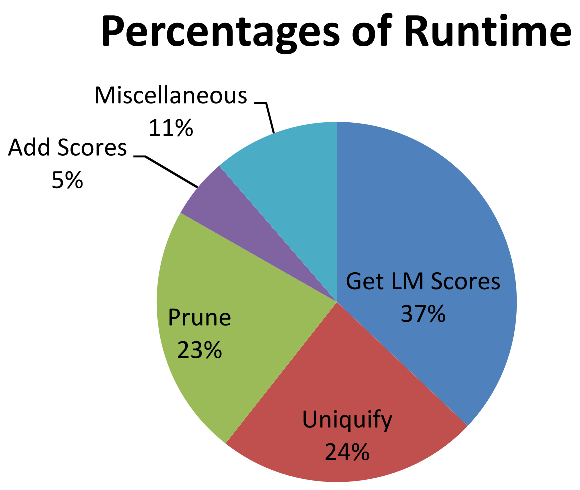


Figure 21: *percentages of runtime of major routines in combining translations in parallel. Notice that the only compute-intensive part, Add Scores (line 14 in Figure 18), takes up only 5% of the overall runtime.*

Parallel CKY parsing does not have an ideal compute-to-memory ratio, either. However, thanks to the limited number of symbols and corresponding rules, its memory accesses are regular and predicable. On the contrary, *Get LM Scores* in parallel CKY-based decoding, which accounts for 37% of the overall runtime, consists of unpredictable memory accesses, since the words that will be examined in the list of translations of a cell cannot be determined prior to the actual computation. Those memory accesses are also irregular when we use compact indexing, since accesses into *First Words*, *Second Words* and *Third Words* are completely random with respect to thread IDs. The same can be said when hash maps are used.

Finally, we can observe from Figure 21 that the maximum speedup we can possibly achieve is 4.3×, which is about twice as the current speedup. *Uniquify* and *Prune* are built from the heavily-optimized Thrust routines, which means we cannot further reduce their runtime. Even if we could manage to accelerate the other parts to a negligible amount of runtime, we are still left with the 47% runtime for *Uniquify* and *Prune*. The theoretically best speedup for our parallel CKY-based MT decoder is thus 2.0× / 47% = 4.3×.

4.4. Related Work

Most of the research effort of the statistical machine translation community has been dedicated to the investigation of various statistical methods to construct language models and translation models with higher translation quality. This leaves researches on efficient decoding algorithms a minority. Furthermore, all of such research work focuses on algorithmic enhancement instead of exploring possibilities of implementation platforms, let alone parallelism. As a result, none of them is directly comparable to our approach.

The most well-known series of research for efficient MT decoding develop the concept of *cube pruning* (Chiang, 2007; Huang and Liang 2007). Cube pruning is essentially a clever way to combine two lists of k translations and get the top k results. They argue that not all k^2 combinations have to be traversed, but we can start from combining the best translations of both lists, and go down the lists in a descending order of probability. The resultant speed improvements often surpass a factor of ten. However, the cube pruning approach is inherently serial and not parallelizable.

Several research projects on efficient MT decoding also stress parallelism, but the parallelism in their work connotes the simultaneous execution of orthogonal MT decoding approaches, instead of a feature of implementation platforms. In Ren and Shi (2002), four subsystems of translation engines are executed independently, whose result with the best quality is then adopted as the final result. Tsukada and Nagata (2004) combine different translation models with weight finite state transducer (WFST) in order to improve translation quality. In this set of work, better translation quality instead of speed improvements is what they aim for.

Perhaps the most relevant research is Li and Khudanpur (2008). They designed and built a scalable MT decoder using cube pruning, beam search and other optimization techniques. They implemented the decoder on a cluster of multicore machine with both distributed and parallel programming. The resultant Java implementation is more than 30 times faster than the original Python implementation. However, it seems that in Li and Khudanpur (2008), the inherent speed difference between Java and Python programs is not credited. Furthermore, their system parallelized upon sentences by making different cores/threads handle different sets of sentences, whereas our approach exploited the parallelism inside a sentence. The comparison between inter-sentence parallelism and intra-sentence parallelism is therefore not meaningful.

As inspired by the coarse-to-fine approaches in CKY parsing (Charniak and Johnson, 2005; Petrov and Klein, 2007b), Petrov et al. (2008) designed a coarse-to-fine syntactic MT system, where coarse translation models as well as coarse language models are derived and utilized to constrain and prune the search space of possible translations before applying finer models. This approach demonstrated a 50-fold speedup over the one-model approach. In principle, this approach is orthogonal to our parallelization approach, and can be superimposed to yield greater speed improvements.

5. Conclusions

In this report, we explored the design spaces of parallelizing two major natural language processing applications, natural language parsing and statistical machine translation. In natural language parsing, we focused on parallelizing the CKY parsing algorithm, which is prevalent 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.

In statistical machine translation, we zeroed in on the possibility of parallelizing the CKY-based decoding algorithm for phrase-based machine translation, the most widely-used machine translation scheme. We proposed several optimization approaches to expose the inherent parallelism and to reduce memory accesses, which are tailor-designed for NVIDIA GPUs. However, the best implementation on GTX480 only runs twice as fast as the optimized sequential C implementation. A detailed runtime and profiling analysis shows that the CKY-based decoding algorithm is memory-intensive and incurs a considerable amount of irregular memory accesses, both of which are harmful to the performance of GPU programs.

6. Acknowledgments

Thank you to Youngmin Yi for your collaboration and dedication in the parallelization of natural language parsing, and Ekaterina Gonina for your collaboration and enlightening discussions in the parallelization of machine translation. I am also grateful for Slav Petrov's generous assistance in utilizing Berkeley Parser. This work was supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery

(Award #DIG07-10227). This work was also supported in part by the 2010 Research Fund of the University of Seoul. Additional supports come from UC Berkeley Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

References

D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, and M. Mitzenmacher. 2009. Real-time Parallel Hashing on the GPU. In *ACM Transactions on Graphics*, 28(5).

Apple - iPhone 4S - Ask Siri to help you get things done. 2011. Retrieved April 10th, 2012, from http://www.apple.com/iphone/features/siri.html.

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. 2003. Accelerating the CKY Parsing Using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):803-810.

X. Carreras, M. Collins, and T. Koo. 2008. TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-Rich Parsing. In *Proceedings of the 12th Conference on Natural Language Learning(CoNLL),* pages 9-16.

B. Catanzaro, B.-Y. Su, N. Sundaram, Y. Lee, M. Murphy, K. Keutzer. 2009. Efficient, High-Quality Image Contour Detection. In *Proceedings of International Conference on Computer Vision (ICCV)*, pages 2381-2388.

E. Charniak and M. Johnson. 2005. Coarse-to-Fine N-Best Parsing and MaxEnt Discriminative Reranking. In *Proceedings of the 43rd Annual Meeting of the Association of Computational Linguistics (ACL)*.

E. Charniak. 2000. A Maximum-Entropy-Inspired Parser. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics (NAACL),* pages 132-139.

D. Chiang. 2005. A Hierarchical Phrase-based Model for Statistical Machine Translation. In *Proceedings of the 43rd Annual Meeting of the Association of Computational Linguistics (ACL)*.

D. Chiang. 2007. Hierarchical Phrase-based Translation. In *Computational Linguistics*, 33(2):201-228.

J. Chong, Y. Yi, N. R. S. A. Faria, and K. Keutzer. 2008. Data-parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors. In *Proceedings of International Workshop on Emerging Applications and Manycore Architectures*.

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.

L. Dagun and R. Menon. 1998. OpenMP: an Industry Standard API for Shared-memory Programming. In *IEEE Computational Science and Engineering*, Vol. 5, No. 1, pages 46-55

A. Dunlop, N. Bodenstab, and B. Roark. 2011. Efficient Matrix-Encoded Grammars and Low Latency Parallelization Strategies for CYK. In *Proceedings of the 12th International Conference on Parsing Technologies(IWPT)*, pages 163-174.

Fermi: NVIDIA's Next Generation CUDA Compute Architecture. 2009. Retrieved April 10th, 2012, from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondet, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty. 2010. Building Watson: An Overview of the DeepQA Project. In *AI Magazine,* 31(3):59-79.

J. Finkel, A. Kleeman, and C. D. Manning. 2008. Efficient, Feature-Based, Conditional Random Field Parsing. In *Proceedings of the 46th Annual Meeting* *of the Association for Computational Linguistics: Human Language Technologies(ACL/HLT)*.

E. P. Giachin and C. Rullent. 1989. A Parallel Parser for Spoken Natural Language. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence(IJCAI)*, pages 1537-1542.

J. Goodman. 1997. Global Thresholding and Multiple-Pass Parsing. In *Proceedings of the 2nd Conference on Empirical Methods in Natural Language Processing(EMNLP)*.

J. Hoberock and N. Bell. 2010. Thrust: A Parallel Template Library. Retrieved April 13th, 2012, from http://www.meganewtons.com/.

L. Huang and D. Chiang. 2005. Better *k*-Best Parsing. In *Proceedings of the 9th International Conference on Parsing Technologies(IWPT),* pages 53-64.

L. Huang and D. Chiang. 2007. Forest Rescoring: Faster Decoding with Integrated Language Models. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 144-151.

IBM - What is Watson? 2011. Retrieved April 10th, 2012, from http://www-03.ibm.com/innovation/us/watson/what-is-watson/index.html .

J. Jackson. 2011. IBM Watson Vanquishes Human Jeopardy Foes. In *PCWorld*. Retrieved April 10th, 2012, from http://www.pcworld.com/businesscenter/article/219893/ibm_watson_vanquishes_human_jeopardy_foes.html.

T. Kasami. 1965. An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab.

D. B. Kirk and W.-M. 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 *Proceedings of the 4th North American Chapter of the Association for*

Computational Linguistics(NAACL), pages 40-47.

P. Koehn. 2004. Pharaoh: a Beam Search Decoder for Phrase-Based Statistical Machine Translation Models. In *Proceedings of the 6th conference of the Association for Machine Translation in the Americas (AMTA)*.

P. Koehn. 2005. EuroParl: A Parallel Corpus for Statistical Machine Translation. In *Proceedings of Machine Translation Summit, 2005*.

P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. 2007. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the Annual Conference of the Association for Computational Linguistics, 2007(ACL)*.

P. Koehn, F. J. Och, and D. Marcu. 2003. Statistical Phrase-based Translation. In *Proceedings of the 4th North American Chapter of the Association for Computational Linguistics(NAACL)*, pages 48-54.

Z. Li and S. Khudanpur. 2008. A Scalable Decoder for Parsing-based Machine Translation with Equivalent Language Model State Maintenance. In *Proceedings of the 2nd ACL Workshop on Syntax and Structure in Statistical Translation (SSST)*, pages 10-18.

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,* 19(2).

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

Proceedings of the 10th Conference of the Pacific Association for Computation Linguistics(PACLING), pages 223-231.

K. Papineni, S. Ruokos, T. Ward, and W.-J. Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311-318.

A. Pauls and D. Klein. 2009. Hierarchical Search for Parsing. In *Proceedings of the 10th North American Chapter of the Association for Computational Linguistics(NAACL)*, pages 557-565.

S. Petrov, A. Haghighi and D. Klein. 2008. Coarse-to-fine Syntactic Machine Translation using Language Projections. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 108-116.

S. Petrov and D. Klein. 2007a. Discriminative Log-linear Grammars with Latent Variables. In *Proceedings of the 20th Annual Meeting of Neural Information Processing Foundation (NIPS)*.

S. Petrov and D. Klein. 2007b. Improved Inference for Unlexicalized Parsing. In *Proceedings of the 8th North American Chapter of the Association for Computational Linguistics(NAACL)*, pages 404-411.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. 2006. Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proceedings of the 44th Annual Meeting of the Association of Computational Linguistics(ACL)*, pages 433-440.

E. Pontelli, G. Gupta, J. Wiebe, and D. Farwell. 1998. Natural Language Processing: A Case Study. In *Proceedings of the 15th National Conference on Artificial Intelligence(AAAI)*.

F. Ren and H. Shi. 2001. Parallel Machine Translation: Principles and Practice. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 2-49.

A. Stolcke. 2002. SRILM - An Extensible Language Modeling Toolkit. In *Proceedings of the International Conference on Spoken Language Processing*, vol. 2, pages 901-904.

C. Tillmann. 2003. A Projection Extension Algorithm for Statistical Machine Translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing(EMNLP)*, pages 1-8.

H. Tsukada and M. Nagata. 2004. Efficient Decoding for Statistical Machine Translation with a Fully Expanded WFST Model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing(EMNLP)*, pages 427-433.

M. P. van Lohuizen. 1999. Parallel Processing of Natural Language Parsers. In *Proceedings of the 15th Conference of Parallel Computing*, pages 17-20.

K. Yamada and K. Knight. 2002. A Decoder for Syntax-based Statistical MT. In *Proceedings of the 40th Annual Meeting of the Association for Computation Linguistics (ACL)*, pages 303-310.

D. H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10.