Sparser, Better, Faster GPU Parsing

David Hall **Taylor Berg-Kirkpatrick** **John Canny** **Dan Klein**
Computer Science Division
University of California, Berkeley
{dlwh,tberg,jfc,klein}@cs.berkeley.edu

Abstract

Due to their origin in computer graphics, graphics processing units (GPUs) are highly optimized for dense problems, where the exact same operation is applied repeatedly to all data points. Natural language processing algorithms, on the other hand, are traditionally constructed in ways that exploit structural sparsity. Recently, Canny et al. (2013) presented an approach to GPU parsing that sacrifices traditional sparsity in exchange for raw computational power, obtaining a system that can compute Viterbi parses for a high-quality grammar at about 164 sentences per second on a mid-range GPU. In this work, we reintroduce sparsity to GPU parsing by adapting a coarse-to-fine pruning approach to the constraints of a GPU. The resulting system is capable of computing over 404 Viterbi parses per second—more than a 2x speedup—on the same hardware. Moreover, our approach allows us to efficiently implement less GPU-friendly minimum Bayes risk inference, improving throughput for this more accurate algorithm from only 32 sentences per second unpruned to over 190 sentences per second using pruning—nearly a 6x speedup.

1 Introduction

Because NLP models typically treat sentences independently, NLP problems have long been seen as "embarrassingly parallel" – large corpora can be processed arbitrarily fast by simply sending different sentences to different machines. However, recent trends in computer architecture, particularly the development of powerful "general purpose" GPUs, have changed the landscape even for problems that parallelize at the sentence level. First, classic single-core processors and main memory architectures are no longer getting substantially faster over time, so speed gains must now come from parallelism within a single machine. Second, compared to CPUs, GPUs devote a much larger fraction of their computational power to actual arithmetic. Since tasks like parsing boil down to repeated read-multiply-write loops, GPUs should be many times more efficient in time, power, or cost. The challenge is that GPUs are not a good fit for the kinds of sparse computations that most current CPU-based NLP algorithms rely on.

Recently, Canny et al. (2013) proposed a GPU implementation of a constituency parser that sacrifices all sparsity in exchange for the sheer horsepower that GPUs can provide. Their system uses a grammar based on the Berkeley parser (Petrov and Klein, 2007) (which is particularly amenable to GPU processing), "compiling" the grammar into a sequence of GPU kernels that are applied densely to every item in the parse chart. Together these kernels implement the Viterbi inside algorithm. On a mid-range GPU, their system can compute Viterbi derivations at 164 sentences per second on sentences of length 40 or less (see timing details below).

In this paper, we develop algorithms that can exploit sparsity on a GPU by adapting coarse-to-fine pruning to a GPU setting. On a CPU, pruning methods can give speedups of up to 100x. Such extreme speedups over a dense GPU baseline currently seem unlikely because fine-grained sparsity appears to be directly at odds with dense parallelism. However, in this paper, we present a system that finds a middle ground, where some level of sparsity can be maintained without losing the parallelism of the GPU. We use a coarse-to-fine approach as in Petrov and Klein (2007), but with only one coarse pass. Figure 1 shows an overview of the approach: we first parse densely with a coarse grammar and then parse sparsely with the fine grammar, skipping symbols that the coarse pass deemed sufficiently unlikely. Using this approach, we see a gain of more than 2x over the dense GPU implementation, resulting in overall speeds of up to 404 sentences per second. For comparison, the publicly available CPU implementation of Petrov and Klein (2007) parses approximately 7 sentences per second per core on a modern CPU.

A further drawback of the dense approach in Canny et al. (2013) is that it only computes Viterbi parses. As with other grammars with a parse/derivation distinction, the grammars of Petrov and Klein (2007) only achieve their full accuracy using minimum-Bayes-risk parsing, with improvements of over 1.5 F1 over best-derivation Viterbi parsing on the Penn Treebank (Marcus et al., 1993). To that end, we extend our coarse-to-fine GPU approach to computing marginals, along the way proposing a new way to exploit the coarse pass to avoid expensive log-domain computations in the fine pass. We then implement minimum-Bayes-risk parsing via the max recall algorithm of Goodman (1996). Without the coarse pass, the dense marginal computation is not efficient on a GPU, processing only 32 sentences per second. However, our approach allows us to process over 190 sentences per second, almost a 6x speedup.

2 A Note on Experiments

We build up our approach incrementally, with experiments interspersed throughout the paper, and summarized in Tables 1 and 2. In this paper, we focus our attention on current-generation NVIDIA GPUs. Many of the ideas described here apply to other GPUs (such as those from AMD), but some specifics will differ. All experiments are run with an NVIDIA GeForce GTX 680, a mid-range GPU that costs around $500 at time of writing. Unless otherwise noted, all experiments are conducted on sentences of length ≤ 40 words, and we estimate times based on batches of 20K sentences.[1] We should note that our experimental condition differs from that of Canny et al. (2013): they evaluate on sentences of length ≤ 30. Furthermore, they use two NVIDIA GeForce GTX *690*s—each of which is essentially a repackaging of two 680s—meaning that our system and experiments would run approximately four times faster on their hardware. (This expected 4x factor is empirically consistent with the result of running their system on our hardware.)

3 Sparsity and CPUs

One successful approach for speeding up constituency parsers has been to use coarse-to-fine inference (Charniak et al., 2006). In coarse-to-fine inference, we have a sequence of increasingly complex grammars G_ℓ. Typically, each successive grammar G_ℓ is a *refinement* of the preceding grammar $G_{\ell-1}$. That is, for each symbol A_x in the fine grammar, there is some symbol A in the coarse grammar. For instance, in a latent variable parser, the coarse grammar would have symbols like NP, VP, etc., and the fine pass would have refined symbols NP_0, NP_1, VP_4, and so on.

In coarse-to-fine inference, one applies the grammars in sequence, computing inside and outside scores. Next, one computes (max) marginals for every labeled span (A, i, j) in a sentence. These max marginals are used to compute a *pruning mask* for every span (i, j). This mask is the set of symbols allowed for that span. Then, in the next pass, one only processes rules that are licensed by the pruning mask computed at the previous level.

This approach works because a low quality coarse grammar can still reliably be used to prune many symbols from the fine chart without loss of accuracy. Petrov and Klein (2007) found that over 98% of symbols can be pruned from typical charts using a simple X-bar grammar without any loss of accuracy. Thus, the vast majority of rules can be skipped, and therefore most computation can be avoided. It is worth pointing out that although 98% of labeled spans can be skipped due to X-bar pruning, we found that only about 79% of binary rule applications can be skipped, because the unpruned symbols tend to be the ones with a larger grammar footprint.

4 GPU Architectures

Unfortunately, the standard coarse-to-fine approach does not naïvely translate to GPU architectures. GPUs work by executing thousands of threads at once, but impose the constraint that large blocks of threads must be executing the same

[1]The implementation of Canny et al. (2013) cannot handle batches so large, and so we tested it on batches of 1200 sentences. Our reimplementation is approximately the same speed for the same batch sizes. For batches of 20K sentences, we used sentences from the training set. We verified that there was no significant difference in speed for sentences from the training set and from the test set.

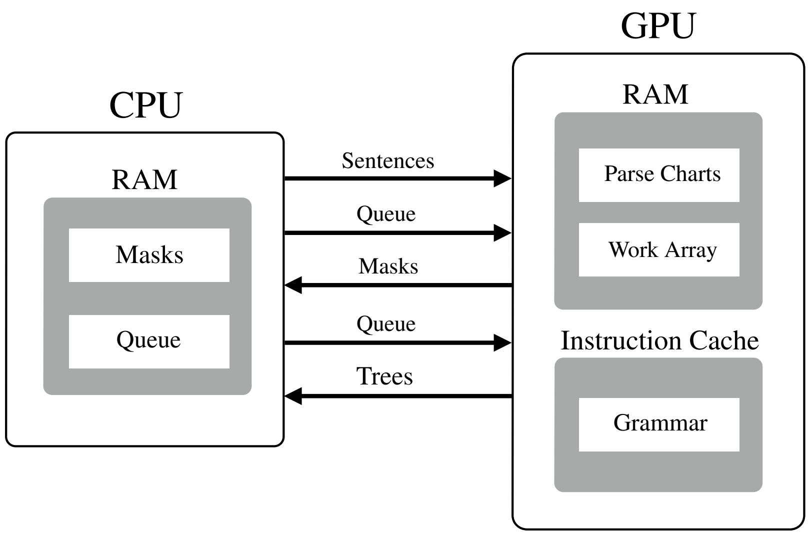

Figure 1: Overview of the architecture of our system, which is an extension of Canny et al. (2013)'s system. The GPU and CPU communicate via a work queue, which ferries parse items from the CPU to the GPU. Our system uses a coarse-to-fine approach, where the coarse pass computes a pruning mask that is used by the CPU when deciding which items to queue during the fine pass. The original system of Canny et al. (2013) only used the fine pass, with no pruning.

instructions in lockstep, differing only in their input data. Thus sparsely skipping rules and symbols will not save any work. Indeed, it may actually slow the system down. In this section, we provide an overview of GPU architectures, focusing on the details that are relevant to building an efficient parser.

The large number of threads that a GPU executes are packaged into blocks of 32 threads called *warps*. All threads in a warp must execute the same instruction at every clock cycle: if one thread takes a branch the others do not, then *all* threads in the warp must follow both code paths. This situation is called *warp divergence*. Because all threads execute all code paths that any thread takes, time can only be saved if an entire warp agrees to skip any particular branch.

NVIDIA GPUs have 8-15 processors called *streaming multi-processors* or SMs.[2] Each SM can process up to 48 different warps at a time: it interleaves the execution of each warp, so that when one warp is stalled another warp can execute. Unlike threads within a single warp, the 48 warps do not have to execute the same instructions. However, the memory architecture is such that they will be faster if they access related memory locations.

[2]Older hardware (600 series or older) has 8 SMs. Newer hardware has more.

A further consideration is that the number of registers available to a thread in a warp is rather limited compared to a CPU. On the 600 series, maximum occupancy can only be achieved if each thread uses at most 63 registers (Nvidia, 2008).[3] Registers are many times faster than variables located in thread-local memory, which is actually the same speed as global memory.

5 Anatomy of a Dense GPU Parser

This architecture environment puts very different constraints on parsing algorithms from a CPU environment. Canny et al. (2013) proposed an implementation of a PCFG parser that sacrifices standard sparse methods like coarse-to-fine pruning, focusing instead on maximizing the instruction and memory throughput of the parser. They assume that they are parsing many sentences at once, with throughput being more important than latency. In this section, we describe their dense algorithm, which we take as the baseline for our work; we present it in a way that sets up the changes to follow.

At the top level, the CPU and GPU communicate via a *work queue* of parse items of the form (s, i, k, j), where s is an identifier of a sentence, i is the start of a span, k is the split point, and j

[3]A thread can use more registers than this, but the full complement of 48 warps cannot execute if too many are used.

Clustering	Pruning	Sent/Sec	Speedup
Canny et al.	–	164.0	–
Reimpl	–	192.9	1.0x
Reimpl	Empty, Coarse	185.5	0.96x
Reimpl	Labeled, Coarse	187.5	0.97x
Parent	–	158.6	0.82x
Parent	Labeled, Coarse	278.9	1.4x
Parent	Labeled, 1-split	**404.7**	**2.1x**
Parent	Labeled, 2-split	343.6	1.8x

Table 1: Performance numbers for computing Viterbi inside charts on 20,000 sentences of length ≤40 from the Penn Treebank. All times are measured on an NVIDIA GeForce GTX 680. ‘Reimpl’ is our reimplementation of their approach. Speedups are measured in reference to this reimplementation. See Section 7 for discussion of the clustering algorithms and Section 6 for a description of the pruning methods. The Canny et al. (2013) system is benchmarked on a batch size of 1200 sentences, the others on 20,000.

is the end point. The GPU takes large numbers of parse items and applies the *entire* grammar to them in parallel. These parse items are enqueued in order of increasing span size, blocking until all items of a given length are complete. This approach is diagrammed in Figure 2.

Because all rules are applied to all parse items, all threads are executing the same sequence of instructions. Thus, there is no concern of warp divergence.

5.1 Grammar Compilation

One important feature of Canny et al. (2013)’s system is *grammar compilation*. Because registers are so much faster than thread-local memory, it is critical to keep as many variables in registers as possible. One way to accomplish this is to unroll loops at compilation time. Therefore, they inlined the iteration over the grammar directly into the GPU kernels (i.e. the code itself), which allows the compiler to more effectively use all of its registers.

However, register space is limited on GPUs. Because the Berkeley grammar is so large, the compiler is not able to efficiently schedule all of the operations in the grammar, resulting in register spills. Canny et al. (2013) found they had to partition the grammar into multiple different kernels. We discuss this partitioning in more detail in Section 7. However, in short, the entire grammar G is broken into multiple clusters G_i where each rule belongs to exactly one cluster.

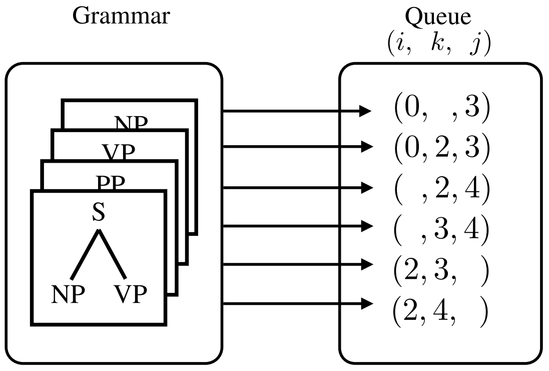

Figure 2: Schematic representation of the work queue used in Canny et al. (2013). The Viterbi inside loop for the grammar is inlined into a kernel. The kernel is applied to all items in the queue in a blockwise manner.

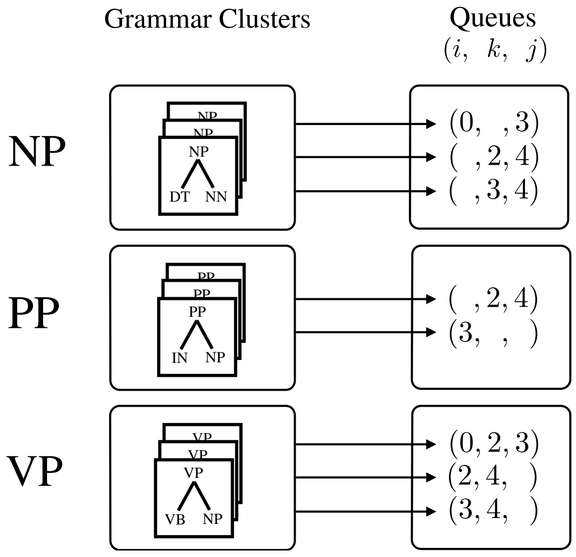

Figure 3: Schematic representation of the work queue and grammar clusters used in the fine pass of our work. Here, the rules of the grammar are clustered by their coarse parent symbol. We then have multiple work queues, with parse items only being enqueued if the span (i, j) allows that symbol in its pruning mask.

All in all, Canny et al. (2013)’s system is able to compute Viterbi charts at 164 sentences per second, for sentences up to length 40. On larger batch sizes, our reimplementation of their approach is able to achieve 193 sentences per second on the same hardware. (See Table 1.)

6 Pruning on a GPU

Now we turn to the algorithmic and architectural changes in our approach. First, consider trying to directly apply the coarse-to-fine method sketched in Section 3 to the dense baseline described above. The natural implementation would be for each thread to check if each rule is licensed before applying it. However, we would only avoid the work of applying the rule if all threads in the warp agreed to skip it. Since each thread in the warp is processing a different span (perhaps even from a different sentence), consensus from all 32 threads on any skip would be unlikely.

Another approach would be to skip enqueuing any parse item (s, i, k, j) where the pruning mask for any of (i, j), (i, k), or (k, j) is entirely empty (i.e. all symbols are pruned in this cell by the coarse grammar). However, our experiments showed that only 40% of parse items are pruned in this manner. Because of the overhead associated with creating pruning masks and the further overhead of GPU communication, we found that this method did not actually produce any time savings at all. The result is a parsing speed of 185.5 sentences per second, as shown in Table 1 on the row labeled 'Reimpl' with 'Empty, Coarse' pruning.

Instead, we take advantage of the partitioned structure of the grammar and organize our computation around the coarse symbol set. Recall that the baseline already partitions the grammar G into rule clusters G_i to improve register sharing. (See Section 7 for more on the baseline clustering.) We create a separate work queue for each partition. We call each such queue a *labeled work queue*, and each one only queues items to which some rule in the corresponding partition applies. We call the set of coarse symbols for a partition (and therefore the corresponding labeled work queue) a *signature*.

During parsing, we only enqueue items (s, i, k, j) to a labeled queue if two conditions are met. First, the span (i, j)'s pruning mask must have a non-empty intersection with the signature of the queue. Second, the pruning mask for the children (i, k) and (k, j) must be non-empty.

Once on the GPU, parse items are processed using the same style of compiled kernel as in Canny et al. (2013). Because the entire partition (though not necessarily the entire grammar) is applied to each item in the queue, we still do not need to worry about warp divergence.

At the top level, our system first computes pruning masks with a coarse grammar. Then it processes the same sentences with the fine grammar. However, to the extent that the signatures are small, items can be selectively queued only to certain queues. This approach is diagrammed in Figure 3.

We tested our new pruning approach using an X-bar grammar as the coarse pass. The resulting speed is 187.5 sentences per second, labeled in Table 1 as row labeled 'Reimpl' with 'Labeled, Coarse' pruning. Unfortunately, this approach again does not produce a speedup relative to our reimplemented baseline. To improve upon this result, we need to consider how the grammar clustering interacts with the coarse pruning phase.

7 Grammar Clustering

Recall that the rules in the grammar are partitioned into a set of clusters, and that these clusters are further divided into subclusters. How can we best cluster and subcluster the grammar so as to maximize performance? A good clustering will group rules together that use the same symbols, since this means fewer memory accesses to read and write scores for symbols. Moreover, we would like the time spent processing each of the subclusters within a cluster to be about the same. We cannot move on to the next cluster until all threads from a cluster are finished, which means that the time a cluster takes is the amount of time taken by the longest-running subcluster. Finally, when pruning, it is best if symbols that have the same coarse projection are clustered together. That way, we are more likely to be able to skip a subcluster, since fewer distinct symbols need to be "off" for a parse item to be skipped in a given subcluster.

Canny et al. (2013) clustered *symbols* of the grammar using a sophisticated spectral clustering algorithm to obtain a permutation of the symbols. Then the rules of the grammar were laid out in a (sparse) three-dimensional tensor, with one dimension representing the parent of the rule, one representing the left child, and one representing the right child. They then split the cube into 6x2x2 contiguous "major cubes," giving a partition of the rules into 24 clusters. They then further subdivided these cubes into 2x2x2 minor cubes, giving 8 subclusters that executed in parallel. Note that the clusters induced by these major and minor cubes need not be of similar sizes; indeed, they often are not. Clustering using this method is labeled 'Reimplementation' in Table 1.

The addition of pruning introduces further considerations. First, we have a coarse grammar, with many fewer rules and symbols. Second, we are able to skip a parse item for an entire cluster if that item's pruning mask does not intersect the cluster's signature. Spreading symbols across clusters may be inefficient: if a parse item licenses a given symbol, we will have to enqueue that item to any queue that has the symbol in its signature, no matter how many other symbols are in that cluster.

Thus, it makes sense to choose a clustering algorithm that exploits the structure introduced by the pruning masks. We use a very simple method: we cluster the rules in the grammar by coarse parent symbol. When coarse symbols are extremely unlikely (and therefore have few corresponding rules), we merge their clusters to avoid the overhead of beginning work on clusters where little work has to be done.[4] In order to subcluster, we divide up rules among subclusters so that each subcluster has the same number of active parent symbols. We found this approach to subclustering worked well in practice.

Clustering using this method is labeled 'Parent' in Table 1. Now, when we use a coarse pruning pass, we are able to parse nearly 280 sentences per second, a 70% increase in parsing performance relative to Canny et al. (2013)'s system, and nearly 50% over our reimplemented baseline.

It turns out that this simple clustering algorithm produces relatively efficient kernels even in the unpruned case. The unpruned Viterbi computations in a fine grammar using the clustering method of Canny et al. (2013) yields a speed of 193 sentences per second, whereas the same computation using coarse parent clustering has a speed of 159 sentences per second. (See Table 1.) This is not as efficient as Canny et al. (2013)'s highly tuned method, but it is still fairly fast, and much simpler to implement.

8 Pruning with Finer Grammars

The coarse to fine pruning approach of Petrov and Klein (2007) employs an X-bar grammar as its first pruning phase, but there is no reason why we cannot begin with a more complex grammar for our initial pass. As Petrov and Klein (2007) have shown, intermediate-sized Berkeley grammars prune many more symbols than the X-bar system. However, they are slower to parse with in a CPU context, and so they begin with an X-bar grammar.

Because of the overhead associated with transferring work items to GPU, using a very small grammar may not be an efficient use of the GPU's computational resources. To that end, we tried computing pruning masks with one-split and two-split Berkeley grammars. The X-bar grammar can compute pruning masks at just over 1000 sentences per second, the 1-split grammar parses 858 sentences per second, and the 2-split grammar parses 526 sentences per second.

Because parsing with these grammars is still quite fast, we tried using them as the coarse pass instead. As shown in Table 1, using a 1-split grammar as a coarse pass allows us to produce over 400 sentences per second, a full 2x improvement over our original system. Conducting a coarse pass with a 2-split grammar is somewhat slower, at a "mere" 343 sentences per second.

9 Minimum Bayes risk parsing

The Viterbi algorithm is a reasonably effective method for parsing. However, many authors have noted that parsers benefit substantially from minimum Bayes risk decoding (Goodman, 1996; Simaan, 2003; Matsuzaki et al., 2005; Titov and Henderson, 2006; Petrov and Klein, 2007). MBR algorithms for parsing do not compute the best derivation, as in Viterbi parsing, but instead the parse tree that maximizes the expected count of some figure of merit. For instance, one might want to maximize the expected number of correct constituents (Goodman, 1996), or the expected rule counts (Simaan, 2003; Petrov and Klein, 2007). MBR parsing has proven especially useful in latent variable grammars. Petrov and Klein (2007) showed that MBR trees substantially improved performance over Viterbi parses for latent variable grammars, earning up to 1.5F1.

Here, we implement the Max Recall algorithm of Goodman (1996). This algorithm maximizes the expected number of correct coarse symbols (A, i, j) with respect to the posterior distribution over parses for a sentence.

This particular MBR algorithm has the advantage that it is relatively straightforward to implement. In essence, we must compute the marginal probability of each fine-labeled span $\mu(A_x, i, j)$, and then marginalize to obtain $\mu(A, i, j)$. Then, for each span (i, j), we find the best possible split

[4]Specifically, after clustering based on the coarse parent symbol, we merge all clusters with less than 300 rules in them into one large cluster.

point k that maximizes $C(i,j) = \mu(A,i,j) + \max_k(C(i,k) + C(k,j))$. Parse extraction is then just a matter of following back pointers from the root, as in the Viterbi algorithm.

9.1 Computing marginal probabilities

The easiest way to compute marginal probabilities is to use the log space semiring rather than the Viterbi semiring, and then to run the inside and outside algorithms as before. We should expect this algorithm to be at least a factor of two slower: the outside pass performs at least as much work as the inside pass. Moreover, it typically has worse memory access patterns, leading to slower performance.

Without pruning, our approach does not handle these log domain computations well at all: we are only able to compute marginals for 32.1 sentences/second, more than a factor of 5 slower than our coarse pass. To begin, log space addition requires significantly more operations than max, which is a primitive operation on GPUs. Beyond the obvious consequence that executing more operations means more time taken, the sheer number of operations becomes too much for the compiler to handle. Because the grammars are compiled into code, the additional operations are all inlined into the kernels, producing much larger kernels. Indeed, in practice the compiler will often hang if we use the same size grammar clusters as we did for Viterbi. In practice, we found there is an effective maximum of 2000 rules per kernel using log sums, while we can use more than 10,000 rules rules in a single kernel with Viterbi.

With coarse pruning, however, we can avoid much of the increased cost associated with log domain computations. Because so many labeled spans are pruned, we are able to skip many of the grammar clusters and thus avoid many of the expensive operations. Using coarse pruning and log domain calculations, our system produces MBR trees at a rate of 130.4 sentences per second, a four-fold increase.

9.2 Scaling with the Coarse Pass

One way to avoid the expense of log domain computations is to use scaled probabilities rather than log probabilities. Scaling is one of the folk techniques that are commonly used in the NLP community, but not generally written about. Recall that floating point numbers are composed of a mantissa m and an exponent e, giving a number $f = m \cdot 2^e$. When a float underflows, the exponent becomes too low to represent the available number of bits. In scaling, floating point numbers are paired with an additional number that extends the exponent. That is, the number is represented as $f' = f \cdot \exp(s)$. Whenever f becomes either too big or too small, the number is rescaled back to a less "dangerous" range by shifting mass from the exponent e to the scaling factor s.

System	Sent/Sec	Speedup
Unpruned Log Sum MBR	32.1	–
Pruned Log Sum MBR	130.4	4.1x
Pruned Scaling MBR	**190.6**	**5.9x**
Pruned Viterbi	404.7	12.6x

Table 2: Performance numbers for computing max constituent (Goodman, 1996) trees on 20,000 sentences of length 40 or less from the Penn Treebank. For convenience, we have copied our pruned Viterbi system's result.

In practice, one scale s is used for an entire span (i,j), and all scores for that span are rescaled in concert. In our GPU system, multiple scores in any given span are being updated at the same time, which makes this dynamic rescaling tricky and expensive, especially since inter-warp communication is fairly limited.

We propose a much simpler static solution that exploits the coarse pass. In the coarse pass, we compute Viterbi inside and outside scores for every span. Because the grammar used in the coarse pass is a projection of the grammar used in the fine pass, these coarse scores correlate reasonably closely with the probabilities computed in the fine pass: If a span has a very high or very low score in the coarse pass, it typically has a similar score in the fine pass. Thus, we can use the coarse pass's inside and outside scores as the scaling values for the fine pass's scores. That is, in addition to computing a pruning mask, in the coarse pass we store the maximum inside and outside score in each span, giving two arrays of scores $s^I_{i,j}$ and $s^O_{i,j}$. Then, when applying rules in the fine pass, each fine inside score over a split span (i,k,j) is scaled to the appropriate $s^I_{i,j}$ by multiplying the score by $\exp\left(s^I_{i,k} + s^I_{k,j} - s^I_{i,j}\right)$, where $s^I_{i,k}, s^I_{k,j}, s^I_{i,j}$ are the scaling factors for the left child, right child, and parent, respectively. The outside scores are scaled analogously.

By itself, this approach works on nearly every sentence. However, scores for approximately

0.5% of sentences overflow (*sic*). Because we are summing instead of maxing scores in the fine pass, the scaling factors computed using max scores are not quite large enough, and so the rescaled inside probabilities grow too large when multiplied together. Most of this difference arises at the leaves, where the lexicon typically has more uncertainty than higher up in the tree. Therefore, in the fine pass, we normalize the inside scores at the leaves to sum to 1.0.[5] Using this slight modification, no sentences from the Treebank under- or overflow.

We know of no reason why this same trick cannot be employed in more traditional parsers, but it is especially useful here: with this static scaling, we can avoid the costly log sums without introducing any additional inter-thread communication, making the kernels much smaller and much faster. Using scaling, we are able to push our parser to 190.6 sentences/second for MBR extraction, just under half the speed of the Viterbi system.

9.3 Parsing Accuracies

It is of course important verify the correctness of our system; one easy way to do so is to examine parsing accuracy, as compared to the original Berkeley parser. We measured parsing accuracy on sentences of length ≤ 40 from section 22 of the Penn Treebank. Our Viterbi parser achieves 89.7 F1, while our MBR parser scores 91.0. These results are nearly identical to the Berkeley parsers most comparable numbers: 89.8 for Viterbi, and 90.9 for their “Max-Rule-Sum” MBR algorithm. These slight differences arise from the usual minor variation in implementation details. In particular, we use one coarse pass instead of several, and a different MBR algorithm. In addition, there are some differences in unary processing.

10 Analyzing System Performance

In this section we attempt to break down how exactly our system is spending its time. We do this in an effort to give a sense of how time is spent during computation on GPUs. These timing numbers are computed using the built-in profiling capabilities of the programming environment. As usual, profiles exhibit an observer effect, where the act of measuring the system changes the execution. Nevertheless, the general trends should more or less be preserved as compared to the unprofiled code.

System	Coarse Pass	Fine Pass
Unpruned Viterbi	–	6.4
Pruned Viterbi	1.2	1.5
Unpruned Logsum MBR	—	28.6
Pruned Scaling MBR	1.2	4.3

Table 3: Time spent in the passes of our different systems, in seconds per 1000 sentences. Pruning refers to using a 1-split grammar for the coarse pass.

To begin, we can compute the number of seconds needed to parse 1000 sentences. (We use seconds per sentence rather than sentences per second because the former measure is additive.) The results are in Table 3. In the case of pruned Viterbi, pruning reduces the amount of time spent in the fine pass by more than 4x, though half of those gains are lost to computing the pruning masks.

In Table 4, we break down the time taken by our system into individual components. As expected, binary rules account for the vast majority of the time in the unpruned Viterbi case, but much less time in the pruned case, with the total time taken for binary rules in the coarse and fine passes taking about 1/5 of the time taken by binaries in the unpruned version. Queueing, which involves copying memory around within the GPU to process the individual parse items, takes a fairly consistent amount of time in all systems. Overhead, which includes transport time between the CPU and GPU and other processing on the CPU, is relatively small for most system configurations. There is greater overhead in the scaling system, because scaling factors are copied to the CPU between the coarse and fine passes.

A final question is: how many sentences per second do we need to process to saturate the GPU’s processing power? We computed Viterbi parses of successive powers of 10, from 1 to 100,000 sentences.[6] In Figure 4, we then plotted the throughput, in terms of number of sentences per second. Throughput increases through parsing 10,000 sentences, and then levels off by the time it reaches 100,000 sentences.

[5]One can instead interpret this approach as changing the scaling factors to $s^{I'}_{i,j} = s^{I}_{i,j} \cdot \prod_{i \leq k < j} \sum_{A} \text{inside}(A, k, k+1)$, where inside is the array of scores for the fine pass.

[6]We replicated the Treebank for the 100,000 sentences pass.

System	Coarse Pass					Fine Pass			
	Binary	Unary	Queueing	Masks	Overhead	Binary	Unary	Queueing	Overhead
Unpruned Viterbi	–	–	–	–	–	5.42	0.14	0.33	0.40
Pruned Viterbi	0.59	0.02	0.19	0.04	0.22	0.56	0.10	0.34	0.22
Pruned Scaling	0.59	0.02	0.19	0.04	0.20	1.74	0.24	0.46	0.84

Table 4: Breakdown of time spent in our different systems, in seconds per 1000 sentences. Binary and Unary refer to spent processing binary rules. Queueing refers to the amount of time used to move memory around within the GPU for processing. Overhead includes all other time, which includes communication between the GPU and the CPU.

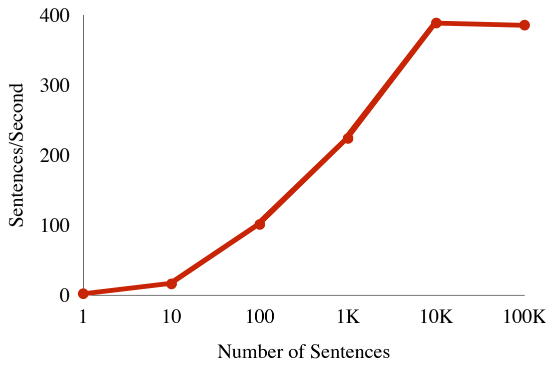

Figure 4: Plot of speeds (sentences / second) for various sizes of input corpora. The full power of the GPU parser is only reached when run on large numbers of sentences.

11 Related Work

Apart from the model of Canny et al. (2013), there have been a few attempts at using GPUs in NLP contexts before. Johnson (2011) and Yi et al. (2011) both had early attempts at porting parsing algorithms to the GPU. However, they did not demonstrate significantly increased speed over a CPU implementation. In machine translation, He et al. (2013) adapted algorithms designed for GPUs in the computational biology literature to speed up on-demand phrase table extraction.

12 Conclusion

GPUs represent a challenging opportunity for natural language processing. By carefully designing within the constraints imposed by the architecture, we have created a parser that can exploit the same kinds of sparsity that have been developed for more traditional architectures.

One of the key remaining challenges going forward is confronting the kind of lexicalized sparsity common in other NLP models. The Berkeley parser's grammars—by virtue of being unlexicalized—can be applied uniformly to all parse items. The bilexical features needed by dependency models and lexicalized constituency models are not directly amenable to acceleration using the techniques we described here. Determining how to efficiently implement these kinds of models is a promising area for new research.

Our system is available as open-source at `https://www.github.com/dlwh/puck`.

Acknowledgments

This work was partially supported by BBN under DARPA contract HR0011-12-C-0014, by a Google PhD fellowship to the first author, and an NSF fellowship to the second. We further gratefully acknowledge a hardware donation by NVIDIA Corporation.

References

John Canny, David Hall, and Dan Klein. 2013. A multi-teraflop constituency parser using GPUs. In *Proceedings of EMNLP*, pages 1898–1907, October.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R Shrivaths, Jeremy Moore, Michael Pozar, et al. 2006. Multilevel coarse-to-fine pcfg parsing. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 168–175. Association for Computational Linguistics.

Joshua Goodman. 1996. Parsing algorithms and metrics. In *ACL*, pages 177–183.

Hua He, Jimmy Lin, and Adam Lopez. 2013. Massively parallel suffix array queries and on-demand phrase extraction for statistical machine translation using gpus. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 325–334, Atlanta, Georgia, June. Association for Computational Linguistics.

Mark Johnson. 2011. Parsing in parallel on multiple cores and gpus. In *Proceedings of the Australasian Language Technology Association Workshop*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *ACL*, pages 75–82, Morristown, NJ, USA.

CUDA Nvidia. 2008. Programming guide.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *NAACL-HLT*.

Khalil Simaan. 2003. On maximizing metrics for syntactic disambiguation. In *Proceedings of IWPT*.

Ivan Titov and James Henderson. 2006. Loss minimization in parse reranking. In *Proceedings of EMNLP*, pages 560–567. Association for Computational Linguistics.

Youngmin Yi, Chao-Yue Lai, Slav Petrov, and Kurt Keutzer. 2011. Efficient parallel cky parsing on gpus. In *Proceedings of the 2011 Conference on Parsing Technologies*, Dublin, Ireland, October.