Efficient Matrix-Encoded Grammars and Low Latency Parallelization Strategies for CYK

Aaron Dunlop, Nathan Bodenstab and Brian Roark
Center for Spoken Language Understanding
Oregon Health & Science University
Portland, OR
[aaron.dunlop,bodenstab,roarkbr]@gmail.com

Abstract

We present a matrix encoding of context-free grammars, motivated by hardware-level efficiency considerations. We find efficiency gains of 2.5–9× for exhaustive inference and approximately 2× for pruned inference, resulting in high-accuracy parsing at over 20 sentences per second. Our grammar encoding allows fine-grained parallelism during chart cell population; we present a controlled study of several methods of parallel parsing, and find near-optimal latency reductions as core-count increases.

1 Introduction

Constituent parsers are important for a number of information extraction subtasks — e.g., anaphora and coreference resolution and semantic role labeling — and parsing time is often a bottleneck for such applications (Bjrne et al., 2010; Banko, 1999). Most constituent parsers leverage the dynamic programming "chart" structure of the CYK algorithm, even when performing approximate inference. The inner loop of the CYK algorithm computes an *argmax* for each constituent span by intersecting the set of observed child categories spanning adjacent substrings with the set of rule productions in the grammar. This 'grammar intersection' operation is the most computationally intensive component of the algorithm. Prior work has shown that the grammar encoding can greatly affect parsing efficiency (c.f Klein and Manning (2001), Moore (2004), Penn and Munteanu (2003)); in this paper we present a matrix encoding that can encode very large grammars to maximize inference efficiency.

This matrix grammar encoding allows a refactoring of the CYK algorithm with two beneficial properties: 1) the number of expensive grammar intersection operations is reduced from $O(n^3)$ to $O(n^2)$; and 2) since grammar intersection is reduced to a set of matrix operations, the resulting algorithm is amenable to fine-grained parallelization.

Most discussion of parallel parsing concentrates on *throughput*, the aggregate number of sentences parsed per second on a particular machine. In this study, we are also interested in applications for which response time is of interest (e.g., real-time speech recognition and machine translation), and thus consider *latency*, the time to parse a single sentence, as a primary objective. Given ideally efficient algorithms and hardware, there would be no tradeoff between the two — that is, we would be able to parallelize each sentence across an arbitrary number of processor cores, reducing latency and increasing throughput linearly with core-count. Unfortunately, hardware constraints and Amdahl's law ensure that we will never achieve that ideal speedup; in practice, we are likely to see some tradeoff. We will demonstrate interesting patterns of the tradeoff between throughput and latency with various parallelization methods, allowing consumers to tailor parsing strategies to particular application requirements.

Our grammar intersection method is amenable to graphics processors (GPUs) and similar massively-parallel architectures. In this work, we perform our analysis on a multicore CPU system. We demonstrate the utility of this approach using a number of different grammars, including the latent variable grammar used by the Berkeley parser (Petrov et al., 2006). We show large speedups compared to a traditional CYK implementation for serial inference, parsing over 20 sentences per second with the Berkeley grammar. Parallelizing this algorithm reduces average latency to .026 seconds.

The remainder of this paper is organized as follows: we begin in Section 2 with background on the CYK algorithm and various general and CYK-specific parallelization considerations. In Section 3 we provide a detailed presentation of our grammar encoding, data structures, and intersection method. In Section 4, we demonstrate their effectiveness on a single core and present controlled experiments comparing several parallelization strategies.

2 Background

2.1 CYK

Algorithm 1 shows pseudocode of the widely used CYK algorithm. Briefly, constituents spanning longer substrings are built from shorter-span constituents via a chart structure, as shown in Figure 1. Span-1 cells (the bottom row of the chart) are initialized with all part-of-speech (POS) tags, and with unary productions spanning a single word. At higher span cells in the chart, such as the dark grey cell in Figure 1, new constituents are built by combining constituents spanning adjacent substrings, guided by the productions in the grammar. With a probabilistic context-free grammar (PCFG) the maximum likelihood solution is found by storing, in each cell in the chart, the highest probability for each category in the non-terminal set V along with a backpointer to where that solution came from. Dynamic programming reduces this potentially exponential search to $O(n^3)$ complexity.

2.2 Parallelism

Since smooth parallelization is one of the benefits of the algorithm we will present in Section 3.2, we begin with background on some of the barriers to efficient parallelism. The overhead of parallelism takes many forms.[1] The operating system consumes processor cycles in thread scheduling; coordination and synchronization of concurrent tasks can leave processors idle; and (more importantly to memory-bound applications such as parsing) context switching between threads often requires flushing the CPU cache, resulting in more memory contention and stalls.

Further, some multi-core architectures share L2 or L3 caches between CPU cores, and nearly all share bandwidth to memory (the ‘front-side bus’, or FSB). Parallel execution threads compete for those resources, and may stall one another. Thus, parallelism can introduce considerable hardware overhead, even if OS- and task-level overhead are minimal, but this impact can be minimized if concurrent threads share common data structures.

2.3 Low-Latency Parallel CYK

We observe that CYK parsing can be parallelized in (at least) three distinct ways, each likely to have different advantages and disadvantages vis-à-vis the bottlenecks just discussed:

Sentence-level: The simplest way to parallelize parsing is to parse sentences or documents independently on separate cores. This approach is well-understood, simple to implement, and quite effective. Total throughput should scale roughly linearly with the number of cores available, at least until we reach the limits of memory bandwidth, but latency is not improved — and may actually increase.

Cell-level: In most forms of CYK iteration, we populate each cell separately, leading to a straightforward form of cell-level parallelism. For example, in bottom-up cell iteration order, we populate one chart row fully before proceeding to the next. The cells on each row are independent of one another, so we can process all cells of a row in parallel. Unfortunately, as we move higher in the chart, there are fewer cells per row, and we must leave CPU cores idle. The highest cells in the chart are often the most densely populated (and require the most processing), an inherent limitation of this form of parallelism.[2]

Ninomiya et al. (1997) explored cell-level parallelization on a 256-processor machine. Their method incurred an overhead of 6–10$\times$ vs. their baseline serial algorithm (depending on sentence length). That is, their parallel algorithm ran 6–10 times slower on a single core than a simpler serial implementation. So even if their approach scaled ideally, many cores would be required to match their serial baseline performance. In practice, their algorithm did not scale linearly and required approximately 64 CPUs to equal their baseline single-CPU performance, and the total speedup observed on 256 CPUs was only 2–4$\times$.

Grammar-level: Parallelization within a chart cell is more difficult to implement, but may avoid some of the weaknesses of the first two methods described. If we can fully parallelize cell population, we can make use of all available cores re-

[1]We are concerned primarily with parallelism within a single machine. Cluster-level parallelism incurs network latency, shared filesystem, and other forms of overhead that do not concern us here.

[2]If optimizing for throughput, those idle threads could be reassigned to subsequent sentences, but cache- and FSB-contention is likely to further increase latency.

Algorithm 1 CYK($w_1 \dots w_n,\ G = (V, T, S^\dagger, P, \rho)$) PCFG G must be in CNF.
α represents the population of the current cell.

```
1: for $t = 1$ to $n$ do                 ▷ span $= 1$ (Words/POS tags)
2:     for $j = 1$ to $|V|$ do
3:         $\alpha_j(t,t) \leftarrow \mathrm{P}(A_j \rightarrow w_t)$
4: for $s = 2$ to $n$ do                 ▷ All spans $> 1$ (rows in the chart)
5:     for $e = s$ to $n$ do              ▷ All end-points (cells in a row)
6:         $b \leftarrow e - s + 1$         ▷ begin-point for cell
7:         $\forall i{\in}V \mid \alpha_i(b,e) \leftarrow argmax_{j,k,m}\mathrm{P}(A_i \rightarrow A_j A_k)\alpha_j(b,m-1)\alpha_k(m,e)$
```

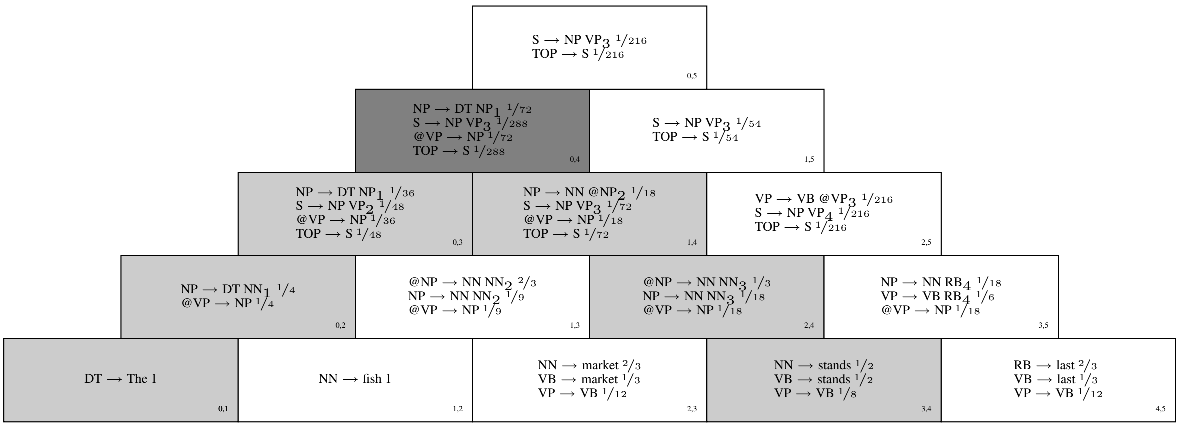

S $\rightarrow$ NP VP$_3$ $^1/_{216}$
TOP $\rightarrow$ S $^1/_{216}$
0,5
NP $\rightarrow$ DT NP$_1$ $^1/_{72}$
S $\rightarrow$ NP VP$_3$ $^1/_{288}$
@VP $\rightarrow$ NP $^1/_{72}$
TOP $\rightarrow$ S $^1/_{288}$
0,4
S $\rightarrow$ NP VP$_3$ $^1/_{54}$
TOP $\rightarrow$ S $^1/_{54}$
1,5
NP $\rightarrow$ DT NP$_1$ $^1/_{36}$
S $\rightarrow$ NP VP$_2$ $^1/_{48}$
@VP $\rightarrow$ NP $^1/_{36}$
TOP $\rightarrow$ S $^1/_{48}$
0,3
NP $\rightarrow$ NN @NP$_2$ $^1/_{18}$
S $\rightarrow$ NP VP$_3$ $^1/_{72}$
@VP $\rightarrow$ NP $^1/_{18}$
TOP $\rightarrow$ S $^1/_{72}$
1,4
VP $\rightarrow$ VB @VP$_3$ $^1/_{216}$
S $\rightarrow$ NP VP$_4$ $^1/_{216}$
TOP $\rightarrow$ S $^1/_{216}$
2,5
NP $\rightarrow$ DT NN$_1$ $^1/_4$
@VP $\rightarrow$ NP $^1/_4$
0,2
@NP $\rightarrow$ NN NN$_2$ $^2/_3$
NP $\rightarrow$ NN NN$_2$ $^1/_9$
@VP $\rightarrow$ NP $^1/_9$
1,3
@NP $\rightarrow$ NN NN$_3$ $^1/_3$
NP $\rightarrow$ NN NN$_3$ $^1/_{18}$
@VP $\rightarrow$ NP $^1/_{18}$
2,4
NP $\rightarrow$ NN RB$_4$ $^1/_{18}$
VP $\rightarrow$ VB RB$_4$ $^1/_6$
@VP $\rightarrow$ NP $^1/_{18}$
3,5
DT $\rightarrow$ The 1
0,1
NN $\rightarrow$ fish 1
1,2
NN $\rightarrow$ market $^2/_3$
VB $\rightarrow$ market $^1/_3$
VP $\rightarrow$ VB $^1/_{12}$
2,3
NN $\rightarrow$ stands $^1/_2$
VB $\rightarrow$ stands $^1/_2$
VP $\rightarrow$ VB $^1/_8$
3,4
RB $\rightarrow$ last $^2/_3$
VB $\rightarrow$ last $^1/_3$
VP $\rightarrow$ VB $^1/_{12}$
4,5

Figure 1: Example CYK chart, with target cell 0,4 highlighted in dark gray and the child cells involved in populating it highlighted in light gray.

gardless of the cell iteration order or the current position in the chart.[3] Because each thread is operating on the same cell, their working sets may align more closely than in other forms of parallelism, reducing context-switch overhead. However, this method implies very fine-grained task divisions and close coordination between threads — when we split a single grammar intersection operation across many threads, each task is quite small. At this fine granularity, locking of shared data structures is impractical, so we must divide tasks such that they share immutable data (the grammar and current cell population) but do not simultaneously mutate the same target data structures (e.g., individual threads may populate separate ranges of non-terminals in the target cell, but must not attempt to populate the same range). Even with careful task division, the task management may overwhelm the potential gains. Youngmin et al. (2011) presented one approach to this problem; we present another approach in Section 3.2.

3 Methods

3.1 Matrix Grammar Encoding

Retrieval of valid grammar rules and their probabilities is an essential component of context-free parsing. High-accuracy grammars can exceed millions of productions, so efficient model access is critical to parsing performance. Prior work on encoding the grammar as a finite state automaton (Klein and Manning, 2001) and prefix compacted tries (Moore, 2004) demonstrated that model encoding can lead to significant efficiency gains in parsing.[4] Motivated to address hardware bottlenecks and constraints, we present a novel encoding in matrix form.

Given a binarized probabilistic context-free grammar (PCFG) defined as the tuple $(V, T, S^\dagger, P, \rho)$ where V is the set of non-terminals, T is the set of terminals, $S^\dagger$ is a special start symbol, P is the set of grammar productions, and ρ is a mapping of grammar productions to probabilities, we subdivide P into binary rules, P_b, and unary rules, P_u.

We encode P_b in matrix form, where the rows of the matrix $1..|V|$ represent a production's left-

[3] Of course, we can utilize sentence-level and cell-level parallelism as well.

[4] All grammar encodings discussed, including our own, only alter efficiency; accuracy remains unchanged.

hand-side non-terminal, and the columns represent a tuple of all possible right-hand-side non-terminals (pairs in a binarized grammar). This forms a matrix of $|V|$ rows and $|V|^2$ columns. Figure 2 shows a simple grammar represented in this format.

In theory, this matrix could contain $|V|^3$ entries, but most grammars of interest are incredibly sparse, populating only a very small fraction of the possible matrix cells. For example, the Berkeley latent-variable grammar defines 1134 non-terminals, so a fully populated binary rule matrix would contain 1.49 billion rules, but the grammar only populates 1.73 million. Thus, we choose a 'compressed sparse column' sparse matrix representation (Tewarson, 1973). This storage structure is quite dense — we store the binary rules of the Berkeley grammar in approximately 10.5 MB of memory. Further, the rules are stored contiguously in memory and in order of access, so the grammar intersection operations should be very cache-efficient.

3.2 Matrix-Vector Grammar Intersection

We now present a novel intersection method, based on the grammar encoding from Section 3.1, which decouples midpoint iteration from grammar intersection, and can reduce the cell population cost considerably. We begin by pointing to Algorithm 1, the standard CYK algorithm. The $argmax$ on line 7 intersects the set of observed child categories spanning adjacent substrings (stored in chart cells) with the set of rule productions found in the grammar. Algorithms 2 and 3 show two possible grammar intersection methods, one which loops over productions in the grammar (Alg. 2) and one which loops over left-children prior to looking for grammar productions (Alg. 3). Song et al. (2008) explored a number of such grammar intersection methods, and found Algorithm 3 to be superior for right-factored grammars. We now present a novel intersection method based on the grammar encoding from Section 3.1. The description in this section is informal, with midpoints omitted for clarity. In Section 3.3, we will formalize the method as an application of a lexicographic semiring.

We represent the population of each chart cell $\boldsymbol{\alpha}$ as a vector in $\mathbb{R}^{|V|}$. Each dimension of this vector represents the (log) probability of a non-terminal

Algorithm 2 Grammar intersection via full grammar loop (backpointer storage omitted). $\alpha(b,e)$ represents the population of the cell spanning words b to e.

```
$\alpha(b,e) \leftarrow 0$
**for** $m = b+1$ to $e-1$ **do**
    **for** $A_i \rightarrow A_j A_k \in P$ **do**
        $x \leftarrow \mathrm{P}(A_i \rightarrow A_j A_k)\alpha_j(b,m-1)\alpha_k(m,e)$
        **if** $x > \alpha_i(b,e)$ **then**
            $\alpha_i(b,e) \leftarrow x$
```

Algorithm 3 Grammar intersection via left child grammar loop

```
$\alpha(b,e) \leftarrow 0$
**for** $m = b+1$ to $e-1$ **do**
    **for** $j \in \alpha(b,m-1)$ **do**
        **for** $A_i \rightarrow A_j A_k \in P$ **do**
            $x \leftarrow \mathrm{P}(A_i \rightarrow A_j A_k)\alpha_j(b,m-1)\alpha_k(m,e)$
            **if** $x > \alpha_i(b,e)$ **then**
                $\alpha_i(b,e) \leftarrow x$
```

in that cell. To perform the $argmax$, we populate a temporary vector **c** of $|V|^2$ dimensions with the cartesian product of all observed non-terminals from the left and right child cells *over all midpoints*. That is, each dimension of this vector represents an ordered pair of non-terminals from the grammar, and its length (score) is the product of the inside probabilities of the respective children. For any child pairs which occur at multiple midpoints, we record only the most probable.

For example, when populating the highlighted cell (0,4) in Figure 1, the first midpoint (m=1) adds (DT,NP), (DT,S), and (DT,@VP) to **c**; the second midpoint (m=2) will add (NP,@NP), (NP,@VP), (@VP,@NP), and so on. If we observe the same pair at multiple midpoints, we retain only the maximum score.

Given a matrix-encoded grammar, G, and the child-cell vector, **c**, we simply multiply G by **c** to produce $\boldsymbol{\alpha}$, the population of the target cell. In Viterbi search, we perform this operation in the $\langle T, T \rangle$ lexicographic semiring, thus computing the maximum probability instead of the sum (described more fully in Section 3.3). Figure 2 demonstrates this Sparse-Matrix $\times$ Vector multiplication (SpMV). The SpMV is the only portion of our algorithm which must access the grammar. We perform that operation once per cell, rather than once per midpoint, reducing the num-

G

	(DT,NP)	(DT,NN)	(NN,NN)	$\cdots$
NP	1/4	1/4	-	
S	-	1/32	1/32	
@VP	-	-	-	
@NP	-	-	1	
$\cdots$				$\cdots$

$\times$

c

Child Pair	Prob
(DT,NP)	1/18
(DT,NN)	0
(NN,NN)	0
$\cdots$	
(NP,VP)	1/72
(DT,S)	1/72
(NP,@NP)	1/12
(NP,NN)	1/72
$\cdots$	$\cdots$

$=$

α

Parent	Prob
NP	1/72
S	1/288
@VP	1/72
@NP	0
$\cdots$	

Figure 2: Example matrix-vector multiplication for cell 0,4 in Figure 1. The grammar G encodes binary rules as a $|V| \times |V|^2$ matrix, with rows representing parents and columns representing child pairs. The vector **c** contains non-terminal child pairs observed across all possible midpoints. The matrix-vector product of $G \times \mathbf{c}$ produces the target cell population, α. Factored categories are prefixed with '@', and backpointers are omitted for clarity.

Algorithm 4 Grammar intersection via Sparse Matrix $\times$ Vector Multiplication. $h(l, r)$ maps $l, r \in V$ to an index of **c**

```
$\mathbf{c} \leftarrow 0$
**for** $m = b + 1$ to $e - 1$ **do**
    **for** $j = 1$ to $|V|$ **do**
        **for** $k = 1$ to $|V|$ **do**
            $i \leftarrow h(\alpha_j(b, m-1), \alpha_k(m, e))$
            **if** $\alpha_j(b, m-1)\alpha_k(m, e) > \mathbf{c}_i$ **then**
                $\mathbf{c}_i \leftarrow \alpha_j(b, m-1)\alpha_k(m, e)$
$\alpha(b, e) \leftarrow G \cdot \mathbf{c}$
```

ber of expensive grammar operations from $O(n^3)$ to $O(n^2)$.

We note the similarity to the formalisms of Valiant (1975), which transforms parsing into boolean matrix multiplication, and Lee (1997), which inverts that transformation. However, the similarity is only superficial; Valient's algorithm populates an upper-triangular matrix, the elements of which are equivalent to CYK chart cells. Each matrix element is a subset of V, the observed population of the analogous chart cell. The matrix is populated by a transitive closure operation, which takes the place of the CYK algorithm. Our matrix operation, on the other hand, is concerned with the population of individual chart cells, the operation accomplished by Valient's $*$ operator.

Decoupling the midpoint iteration from grammar intersection is not contingent on our matrix-vector encoding. The optimization in Graham et al. (1980) also refactors the CYK algorithm to result in $O(n^2)$ grammar intersection operations by changing the dynamic programming to iterate through right (or left) child cells and build new (parent) categories in multiple chart cells at once. Similarly, the grammar-loop intersection of Algorithm 2 could be modified to first maximize over all midpoints, then iterate over grammar productions as is done in Algorithm 4. However, neither variation lends itself to straightforward parallelization, and the required synchronization would severely impact parallel efficiency.

In contrast, the cartesian product and matrix-vector operations of our SpMV method parallelize easily across many cores. We subdivide V into segments, one for each thread. Each thread iterates over its own subset of V in the left child cell and combines with all entries in the right child cell, populating an entry in **c** for each observed child pair. **c** is represented as independent segments safe for lock-free mutation by independent threads (see Section 3.4).

To perform the matrix-vector operation in parallel, we retain the same segments of **c**, and segment G similarly. Each thread t multiplies its segment $G_t \cdot \mathbf{c_t}$, producing a vector $\alpha_\mathbf{t}$. We then merge the $\alpha_\mathbf{t}$ vectors into the final α. Since $|V| << |\mathbf{c}|$, this final merge is quite efficient.

3.3 Lexicographic Semiring

We now present Algorithm 4 more formally as an application of a lexicographic semiring (Golan, 1999). Roark et al. (2011) recently applied lexicographic semirings to language-model encoding. We will follow their notational conventions, and refer the interested reader to their detailed discussion.

A semiring is a ring, possibly lacking negation, defining two operations $\oplus$ and $\otimes$ and their respective identity elements $\bar{0}$ and $\bar{1}$ (Kuich and Salomaa, 1985). One common example in speech and language applications is the *tropical semiring* $(\mathbb{R} \cup \{\infty\}, min, +, \infty, 0)$. min is the $\oplus$ operation, with identity ∞, and $+$ is the $\otimes$, with identity 0. This definition is often used for Viterbi search, using negative log probabilities as costs.

A lexicographic semiring is defined over tuples of weights $\langle W_1, W_2 \ldots W_n \rangle$, with the condition that the tuples can be ordered first by W_1, then by W_2, and so on (similar to lexicographic string comparison, resulting in the name). We use the $\langle T, T \rangle$ semiring, defined as a pair of tropical weights:

$$\langle w_1, w_2 \rangle \oplus \langle w_3, w_4 \rangle = \begin{cases} \langle w_1, w_2 \rangle & \begin{array}{l} \text{if } w_1 < w_3 \text{ or} \\ (w_1 = w_3 \text{ \&} \\ \ w_2 < w_4) \end{array} \\ \langle w_3, w_4 \rangle & \text{otherwise} \end{cases}$$

$$\langle w_1, w_2 \rangle \otimes \langle w_3, w_4 \rangle = \langle w_1 + w_3, w_2 + w_4 \rangle$$

In our application, W_1 encodes the negative log probability of a production in G or of an observed non-terminal in the chart. W_2 encodes the midpoint of the maximum-probability analysis.[5] To perform grammar intersection using this semiring, we encode the grammar matrix as described in Section 3.2, and include 0 as W_2 for each grammar entry (since this weight is constant, it need not be encoded in the grammar representation).

We populate a vector of tuples $\mathbf{c}_i$ for each possible midpoint of the cell, and $\mathbf{c} = \mathbf{c_1} \oplus \mathbf{c_2} \oplus \ldots \mathbf{c_{span}}$. $\oplus$ compares with min, so the entries in $\mathbf{c}$ will be from the maximum probability midpoints and the first midpoint will 'win' in the case of a tie.

When we multiply $G \cdot \mathbf{c}$ in the $\langle T, T \rangle$ semiring, we use the $\otimes$ multiplication operator on each individual element, and the $\oplus$ addition operator for the sum. Since W_2 is 0 for all entries in G, the midpoints are simply carried over from $\mathbf{c}$, and $G \cdot \mathbf{c}$ is the minimum-cost path to each observed non-terminal.

SpMV optimizations have been explored extensively in the high-performance computing literature (c.f., for example, Williams et al. (2009), Bell and Garland (2009), Goumas et al. (2008)) Although the matrix-vector operations in the $\langle T, T \rangle$ semiring is quite distinct from SpMV in the real semiring, we anticipate that some of those algorithms will apply, and would be of particular interest if parsing with grammars more densely populated than those we explore in this study.

[5]Since W_2 represents a midpoint, we could alter the definition to specify that $W_2 \in \mathbb{N}$, but the standard tropical semiring is adequate and slightly simpler.

3.4 Vector Data Structure

We have already discussed a memory- and cache-efficient encoding of G. We must also represent $\mathbf{c}$ efficiently. Although this data structure is not a primary contribution of this work, we do note that the choice of vector representation greatly impacts overall parsing efficiency, so we will briefly describe our choices.

We represent $\mathbf{c}$ with a perfect hash of the form $h(l, r) \rightarrow [m]$, mapping left and right children to a matrix column. Since a perfect hash function ensures no collisions, this function is reversible ($h^{-1}([m]) \rightarrow (l, r)$), allowing recovery of the left and right children from their hashed representation. We construct $|V|$ different hash functions, mapping $h_i(r) \rightarrow [m_i]$. We store the data structures for these functions adjacently in memory, such that iterating over the entire range of $\mathbf{c}$ accesses memory in roughly linear order.[6] Global optimization of a perfect hash is an NP-complete problem. Even though we only create the hash once during initialization, we want to avoid exponential effort and instead use a displacement heuristic (Tarjan and Yao, 1979) to pack the hash efficiently, achieving 50-80% occupancy for most grammars. We elected not to use a minimal perfect hash, since the decrease in storage space comes at the cost of increased memory access.

4 Evaluation

We compare exhaustive and pruned parsing efficiency with several other competitive parsing implementations. We performed all matrix-encoded parsing and parallelization experiments using the open-source BUBS parser (Bodenstab and Dunlop, 2011). The parser framework is grammar agnostic, permitting experiments on grammars of various sizes, and it implements both exhaustive inference and 'Adaptive Beam Pruning', as described in Bodenstab et al. (2011). For exhaustive parsing, we use BUBS implementation of Algorithms 2 and 3 and Mark Johnson's highly optimized C implementation, `lncky` (Johnson, 2006) as baselines; for pruned inference, we compare with the Charniak parser (Charniak, 2000), the

[6]On most modern CPUs, linear memory access patterns allow aggressive and effective data pre-fetching into cache, avoiding costly CPU stalls.

	Markov-0	Markov-2	Parent	Berkeley SM6
Categories	100	3092	6971	1134
Binarized Rules	3859	13649	25229	1,725,570
F-score	60.7	71.9	77.5	89.3
BUBS Grammar loop (Algorithm 2)	0.16	1.92	23.4	29.9
BUBS Left-child loop (Algorithm 3)	0.13	0.50	0.8	63.0
Johnson (2006)	0.10	0.22	0.3	36.0
SpMV (this paper)	0.04	0.26	1.2	3.2

Table 1: Exhaustive Viterbi parse times (average seconds/sentence, lower is better) over WSJ Section 22 for various grammars. All parsers produce the same maximum-likelihood parse trees.

Berkeley parser (Petrov et al., 2006), and with the aforementioned 'Adaptive Beam Search' system implemented in BUBS. The Charniak parser is written in C and parses with a lexicalized grammar. The BUBS and Berkeley parsers are implemented in Java and parse with a latent-variable grammar.[7]

A brief note about implementation choices is appropriate here. Java has often been viewed as notoriously slow, a perception well-established when Java runtime environments were interpreted rather than compiled. Recent advances in virtual machine technology have largely eliminated the differential between Java and statically-compiled languages such as Fortran and C (c.f. Amedro et al. (2010), Kotzmann et al. (2008), Paleczny et al. (2001), Click et al. (2005), Click et al. (2007), Wrthinger et al. (2007), and Tene et al. (2009)).

We performed all trials on a 12-core Linux machine (2 × Intel® Xeon X5650 CPUs). Each core can execute 2 simultaneous threads, for a total of 24 concurrent threads. For the parsers implemented in Java, we used the Oracle 1.6.0_26 Virtual Machine.

4.1 Exhaustive Serial Search

In Table 1, we present exhaustive search results with four grammars, each induced from the Penn Treebank Sections 2-21 (Marcus et al., 1999). The Markov-order-0 and Markov-order-2 grammars were markovized as described in Manning and Schuetze (1999). The parent-annotated grammar further splits the states of the Markov-order-2 grammar by annotating each non-terminal with its parent category, as described in Johnson (1998). This expands the vocabulary greatly, but the ruleset somewhat less so. The Berkeley grammar (Petrov et al., 2006) is a high-accuracy unlexicalized grammar, learned by iteratively splitting and merging non-terminals. Its vocabulary is relatively small (particularly in comparison with the parent-annotated grammar), but the ruleset is quite large. All grammars examined are right-factored, so we evaluate Algorithm 3, per the trials in Song et al. (2008).

Johnson's C implementation outperforms the default BUBS exhaustive implementations, primarily (we believe) due to BUBS use of memory-inefficient Java objects. Our matrix grammar encoding and SpMV grammar intersection algorithm perform very well in comparison to both baseline systems; for the Markov-order-2 and Parent-annotated grammars, which have large vocabularies and relatively small rulesets, our approach performs similarly to Johnson's C implementation. For the grammars with large rulesets relative to their vocabularies (Markov-order-0 and Berkeley), our approach provides a dramatic speedup — over 9× vs. our fastest baseline using the Berkeley grammar. Other experiments not reported here indicate that the grammar representation accounts for the majority of this speedup, with the grammar intersection method accounting for an additional improvement of approximately 35%. We anticipate that potential users will primarily be interested in high-accuracy grammars, particularly for non-exact inference, so we focus all other empirical trials on the Berkeley grammar.

4.2 Pruned Serial Search

Most state-of-the-art context-free parsers resort to approximate inference techniques to decode efficiently. These methods include Coarse-to-Fine (Petrov et al., 2006), A* (Klein and Manning, 2003; Pauls et al., 2010), best-first (Caraballo and Charniak, 1998; Charniak, 2000), and beam

[7]By default, the Berkeley parser marginalizes over the latent-variables in the grammar and retrieves the Max-Rule parse tree; for fair comparison with our approach, we report timings in its simpler Viterbi-search mode.

	F-score	Sent/sec
Charniak (2000)	90.3	1.7
Berkeley (CTF Viterbi)	89.3	4.7
Adaptive Beam w/Alg. 3	89.0	10.2
Adaptive Beam w/Alg. 4	89.1	21.9

Table 2: Pruned parse times over WSJ Section 22 (average sentences/sec, higher is better).

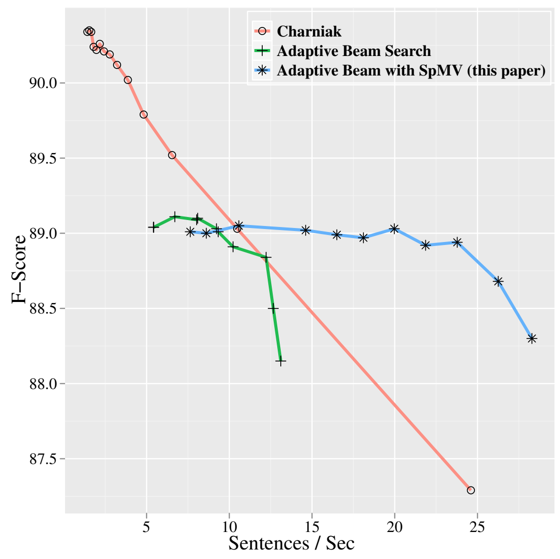


Figure 3: Comparison of F-score vs. speed over a variety of pruning parameterizations for adaptive cell pruning and for our algorithm.

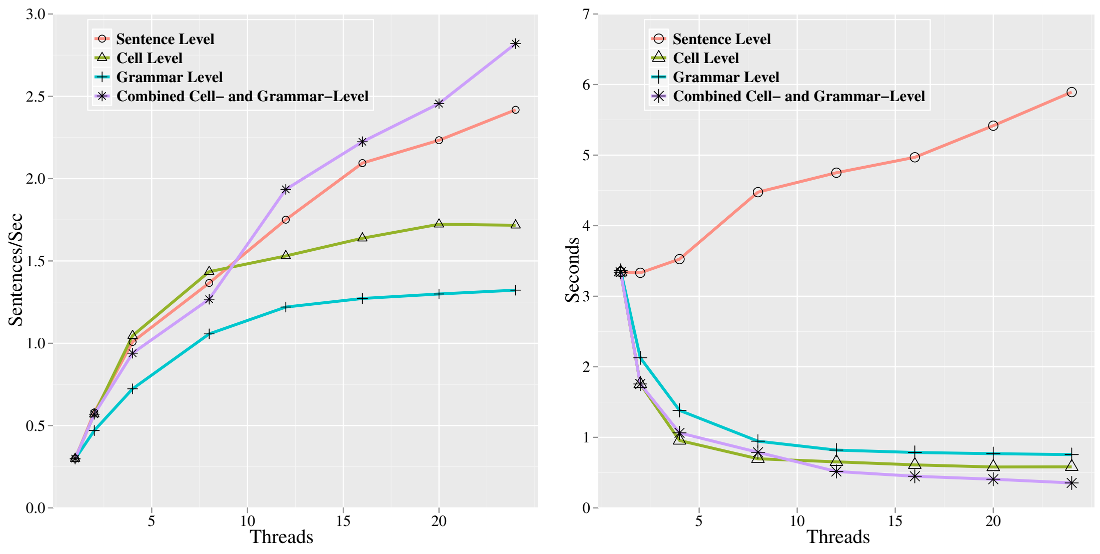
search (Collins, 1999). These methods reduce the search space greatly, allowing effective search in reasonable time, although only A* guarantees finding the globally optimal solution. We take as our primary baseline, the 'Adaptive Beam Search' system implemented in BUBS. This technique predicts the appropriate population of each cell individually, allowing heavier pruning in areas of the chart in which the model predicts less ambiguity. When the lexical context is sufficient to disambiguate constituent structure, it prunes entire cells, and thus generalizes Roark and Hollingshead's linear-time chart constraint algorithm (2009). Our grammar intersection method works well with Adaptive Beam Search. Table 2 shows a speedup of over 2× vs. the baseline implementation, and an even greater advantage vs. other competitive parsers. The Charniak and Adaptive Beam pruning systems both have tunable parameters, controlling their accuracy vs. efficiency operating point. Figure 3 shows empirical results over a range of those tuning parameters for those two implementations and for our approach. For a given parameterization, the search space explored by our approach is identical to that explored by Bodenstab et al., (modulo minor differences in unary processing), so the efficiencies achieved are directly comparable. We find consistently improved speed across all pruning thresholds.

4.3 Exhaustive Parallel Search

We now move to evaluating parallelization methods. Our baseline parsers could be parallelized at a sentence-level, and possibly at a cell-level, but having already established dramatic gains vs. those approaches for serial parsing, we will focus all these trials on our own SpMV algorithm — thus, the sentence-level results reported serve as a 'baseline' of sorts, albeit one already demonstrated to be a dramatic improvement on standard baselines. We parallelize the SpMV implementation using the three parallelization strategies discussed in Section 2.3., and compare throughput and latency. To explore potential additive effects, we include a system combining cell-level and grammar-level parallelism, using several grammar-level threads for each cell-level thread, over the same range of total thread count. Note that some serial processing is required for each sentence (primarily initialization of the chart and extraction of the final parse tree), but these operations consume only 1.5% of the total time.

Executed with a single thread, the cell-level and grammar-level parallel implementations incur an overhead of less than 1%, which compares very favorably with the 500–900% overhead in Ninomiya et al. (1997). Figure 4 shows throughput and latency of each parallelization approach as thread count increases. All approaches show improved throughput with increased thread-count. Cell-level and grammar-level approaches begin to level off around 12 threads, when all physical cores are occupied; combining the two appears to benefit further from Hyper-threading, achieving throughput superior to sentence-level threading.

In Figure 4b, we see two interesting effects regarding latency: 1) We expected the sentence-parallel approach to produce fairly constant latency, but instead found that latency jumped considerably after only 4 threads; 2) Row-level and grammar-level approaches show large decreases in latency as thread count is increased, and the combination again shows additive gains — an overall reduction in latency of approximately 9× and an



(a) Throughput, in sentences per second.

(b) Latency, in seconds per sentence.

Figure 4: Exhaustive SpMV search throughput and latency vs. thread-count.

improvement of nearly 40% vs. cell-level threading alone.

While an improvement of $9\times$ is quite impressive, we anticipate that further gains might be possible. We found that both cell-level and grammar-level methods often leave numerous threads idle, and CPU monitoring rarely shows all cores being occupied. Our observations lead us to believe that much of the 'lost' processor time is going to the task handling and inter-thread communication, and we are optimistic that hardware threading will extend the gains observed for these methods.

4.4 Pruned Parallel Search

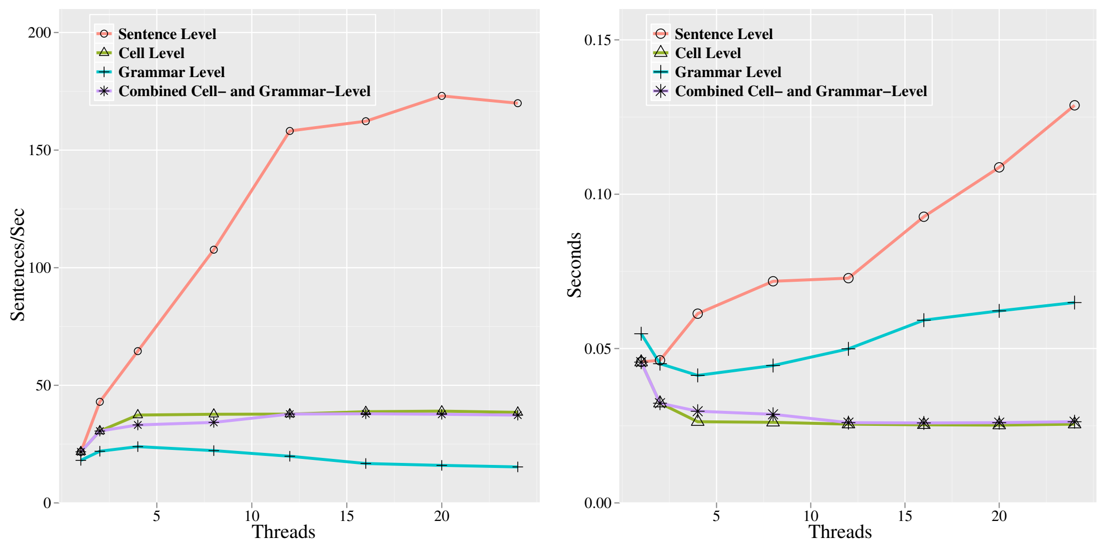
Figure 5 presents similar trials for pruned parallel search (once again, all trials use our grammar intersection method). In this case, we find that the cell-level and combined approaches perform quite strongly — nearly optimally, in fact. In contrast to the relatively small serial portions of exhaustive search, pruned search requires some fairly expensive serial operations. The adaptive cell pruning initialization is quite costly, consuming over 35% of the total time (c.f. Bodenstab et al., 2011). In total, the serial steps account for approximately 45% of the time, so the observed 44% increase in throughput and reduction in latency, although much smaller than that of sentence-level threading, is nearly optimal. We anticipate that parallelizing the pruning initialization should further improve throughput and latency.

For grammar-level parallelism in Figure 5b, however, we find a somewhat counterintuitive result: increasing the thread-count *increases* latency. This is due to the characteristics of the grammar and the severity of pruning during inference. The Berkeley grammar has a small non-terminal set ($|V| = 1134$), but a large ruleset ($|P_b| = 1.7$ million). When performing exhaustive search, many cells are densely populated (average cell population is 450 of 1134). When performing pruned search, the cell populations are naturally much sparser (at most 30 entries, and often fewer). Thus, the grammar-level parallel tasks are much smaller. Task management overhead grows in importance as the task size shrinks, and quickly overwhelms the potential gains of additional execution threads. As already mentioned, hardware thread-management is likely to ameliorate this problem.

5 Discussion and Future Work

We have presented a novel matrix grammar encoding which has several beneficial properties. Access to grammar rules encoded in this manner is very cache-efficient, and it enables cell population using a Sparse Matrix $\times$ Vector grammar intersection. As a well-understood matrix operation, this reduction allows very fine-grained parallelism.

We demonstrated dramatic speedups on exhaustive serial parsing, and a large benefit vs. strong baselines in pruned inference. We found that the efficiency of fine-grained parallel parsing is limited by the thread management overhead, but nevertheless found considerable improvements in



(a) Throughput, in sentences per second.

(b) Latency, in seconds per sentence.

Figure 5: Pruned search throughput and latency vs. thread-count.

throughput and latency, which may be of interest to end-user applications and others with response-time constraints.

This work is available in the open-source BUBS parser for further research or practical application. In future work, we plan to extend our approach to more parallel architectures, including graphics processing units. We also plan to parallelize the adaptive beam width pruning, and we want to explore the SpMV optimizations discussed in Section 3.3.

Acknowledgments

This research was supported in part by NSF Grant #IIS-0811745. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

Brian Amedro, Denis Caromel, Fabrice Huet, Vladimir Bodnartchouk, Christian Delb, and Guillermo L. Taboada. 2010. HPC in java: experiences in implementing the NAS parallel benchmarks. In *Proceedings of the 10th WSEAS international conference on applied informatics and communications and 3rd WSEAS international conference on Biomedical electronics and biomedical informatics*, page 221230.

Michele Banko. 1999. *Open Information Extraction for the Web*. PhD dissertation, University of Washington, Seattle, Washington.

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, Portland, Oregon. ACM.

Jari Bjrne, Filip Ginter, Sampo Pyysalo, Jun'ichi Tsujii, and Tapio Salakoski. 2010. Complex event extraction at PubMed scale. *Bioinformatics*, 26(12):382–390, June.

Nathan Bodenstab and Aaron Dunlop. 2011. BUBS parser. http://code.google.com/p/bubs-parser/.

Nathan Bodenstab, Aaron Dunlop, Brian Roark, and Keith Hall. 2011. Beam-Width prediction for efficient Context-Free parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 440–449, Portland, Oregon, June.

Sharon A Caraballo and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24:275298, June.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st North*

American chapter of the Association for Computational Linguistics conference, pages 132–139, Seattle, Washington. Morgan Kaufmann Publishers Inc.

Cliff Click, Gil Tene, and Michael Wolf. 2005. The pauseless GC algorithm. *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, page 4656.

Clifford N. Click, Christopher A. Vick, and Michael H. Paleczny. 2007. System and method for range check elimination via iteration splitting in a dynamic compiler, May. US Patent 7,222,337.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. PhD dissertation, University of Pennsylvania.

Jonathan Samuel Golan. 1999. *Semirings and their Applications*. Springer, July.

Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2008. Understanding the performance of sparse matrix-vector multiplication. In *PDP08: Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing*.

Susan L. Graham, Michael Harrison Ruzzo, and Walter L. 1980. An improved Context-Free recognizer. *ACM Trans. Program. Lang. Syst.*, 2(3):415–462.

Mark Johnson. 1998. PCFG models of linguistic tree representations. *Comput. Linguist.*, 24(4):613–632.

Mark Johnson. 2006. lncky. http://www.cog.brown.edu/˜mj/Software.htm.

Dan Klein and Christopher D. Manning. 2001. Parsing with treebank grammars: Empirical bounds, theoretical models, and the structure of the penn treebank. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, pages 338–345, Toulouse, France, July.

Dan Klein and Christopher D. Manning. 2003. A* parsing. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL '03)*, pages 40–47, Edmonton, Canada.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mssenbck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the java HotSpot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5:7:1–7:32, May.

Werner Kuich and Arto Salomaa. 1985. *Semirings, Automata, Languages*. EATCS Monographs on Theoretical Computer Science, Number 5. Springer-Verlag, Berlin, Germany.

Lillian Lee. 1997. Fast Context-Free parsing requires fast boolean matrix multiplication. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 9–15, Madrid, Spain, July. ACL.

Christopher D. Manning and Hinrich Schuetze. 1999. *Foundations of Statistical Natural Language Processing*. The MIT Press, June.

Mitchell P Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. 1999. *Treebank-3*. Linguistic Data Consortium, Philadelphia.

Robert C Moore. 2004. Improved left-corner chart parsing for large context-free grammars. *New developments in parsing technology*, page 185201.

Takashi Ninomiya, Kentaro Torisawa, Taura Kinjiro, and Tsujii Jun'ichi. 1997. A parallel CKY parsing algorithm on Large-Scale Distributed-Memory parallel machines. In *PACLING '97*, pages 223–231, Tokyo, Japan.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java hotspot server compiler. *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium*, pages 1–12.

Adam Pauls, Dan Klein, and Chris Quirk. 2010. Top-down k-best a* parsing. In *Proceedings of ACL 2010*, page 200204, Morristown, NJ, USA.

Gerald Penn and Cosmin Munteanu. 2003. A tabulation-based parsing method that reduces copying. In *Proceedings of ACL '03*, pages 200–207, Sapporo, Japan.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia. ACL.

Brian Roark and Kristy Hollingshead. 2009. Linear complexity Context-Free parsing pipelines via chart constraints. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 647–655, Boulder, Colorado, June. ACL.

Brian Roark, Richard Sproat, and Izhak Shafran. 2011. Lexicographic semirings for exact automata encoding of sequence models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, Portland, Oregon, June. ACL.

Xinying Song, Shilin Ding, and Chin-Yew Lin. 2008. Better binarization for the CKY parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 167–176, Honolulu, Hawaii, October. ACL.

Robert Endre Tarjan and Andrew Chi-Chih Yao. 1979. Storing a sparse table. *Commun. ACM*, 22(11):606–611.

Gil Tene, Jack H. Choquette, Scott Sellers, and Clifford N. Click. 2009. Array access, August. US Patent 7,577,801.

Reginald P. Tewarson. 1973. *Sparse Matrices. Mathematics in Science and Engineering Volume 99.* Academic Press, April.

Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–314, April.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, March.

Thomas Wrthinger, Christian Wimmer, and Hanspeter Mssenbck. 2007. Array bounds check elimination for the java HotSpot\ client compiler. *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, page 125133.

Yi Youngmin, Lai Chao-Yue, Slav Patrov, and Kurt Keutzer. 2011. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies*, Dublin, Ireland, October.