Language Identification and Modeling in Specialized Hardware

Kenneth Heafield[*,†] **Rohan Kshirsagar**[*] **Santiago Barona**[*]
[*] Bloomberg L.P.
731 Lexington Ave.
New York, NY 10022 USA
[†] University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB, UK
{kheafield,rkshirsagar2,sbarona}@bloomberg.net

Abstract

We repurpose network security hardware to perform language identification and language modeling tasks. The hardware is a deterministic pushdown transducer since it executes regular expressions and has a stack. One core is 2.4 times as fast at language identification and 1.8 to 6 times as fast at part-of-speech language modeling.

1 Introduction

Larger data sizes and more detailed models have led to adoption of specialized hardware for natural language processing. Graphics processing units (GPUs) are the most common, with applications to neural networks (Oh and Jung, 2004) and parsing (Johnson, 2011). Field-programmable gate arrays (FPGAs) are faster and more customizable, so grammars can be encoded in gates (Ciressan et al., 2000). In this work, we go further down the hardware hierarchy by performing language identification and language modeling tasks on an application-specific integrated circuit designed for network monitoring.

The hardware is programmable with regular expressions and access to a stack. It is therefore a deterministic pushdown transducer. Prior work used the hardware mostly as intended, by scanning hard drive contents against a small set of patterns for digital forensics purposes (Lee et al., 2008). The purposes of this paper are to introduce the natural language processing community to the hardware and evaluate performance.

We chose the related tasks of language identification and language modeling because they do not easily map to regular expressions. Fast language classification is essential to using the web as a corpus (Smith et al., 2013) and packages compete on speed (Lui and Baldwin, 2012). Extensive literature on fast language models comprises a strong baseline (Stolcke, 2002; Federico et al., 2008; Heafield, 2011; Yasuhara et al., 2013). In both cases, matches are frequent, which differs from network security and forensics applications where matches are rare.

2 Related Work

Automata have been emulated on CPUs with AT&T FSM (Mohri et al., 2000) and OpenFST (Allauzen et al., 2007), on GPUs (Rudomín et al., 2005; He et al., 2015), and on FPGAs (Sidhu and Prasanna, 2001; Lin et al., 2006; Korenek, 2010). These are candidates for the ASIC we use. In particular, gappy pattern matching (He et al., 2015) maps directly to regular expressions.

GPUs have recently been applied to the related problem of parsing (Johnson, 2011; Yi et al., 2011). These operate largely by turning a sparse parsing problem into a highly-parallel dense problem (Canny et al., 2013) and by clustering similar workloads (Hall et al., 2014). Since the hardware used in this paper is a deterministic pushdown automaton, parsing ambiguous natural language is theoretically impossible without using the CPU as an oracle. Hall et al. (2014) rely on communication between the CPU and GPU, albeit for efficiency reasons rather than out of necessity.

Work on efficiently querying backoff language models (Katz, 1987) has diverged from a finite state representation. DALM (Yasuhara et al., 2013) is an efficient trie-based representation using double arrays while KenLM (Heafield, 2011) has traditional tries and a linear probing hash table. We use the fastest baselines from both.

3 Programming Model

The fundamental programming unit is a POSIX regular expression including repetition, line boundaries, and trailing context. For example, `a[bc]` matches “ab” and “ac”.

When an expression matches, the hardware can output a constant to the CPU, output the span matched, push a symbol onto the stack, pop from the stack, or halt. There is little meaning to the order in which the expressions appear in the program. All expressions are able to match at any time, but can condition on the top of the stack. This is similar to the `flex` tool (Lesk and Schmidt, 1975), which refers to stack symbols as start conditions.

4 Language Identification

We exactly replicate the model of `langid.py` (Lui and Baldwin, 2012) to identify 97 languages. Their Naïve Bayes model has 7,480 features f_i, each of which is a string of up to four bytes (Lui and Baldwin, 2011). Inference amounts to collecting the count c_i of each feature and computing the most likely language l given model p.

$$l^* = \operatorname*{argmax}_l p(l) \prod_i p(f_i|l)^{c_i}$$

We use the hardware to find all instances of features in the input. Feature strings are converted to literal regular expressions. When the hardware matches the expression for feature f_i, it outputs the unique feature index i. Since the hardware has no user-accessible arithmetic, the CPU accumulates feature counts c_i in an array and performs subsequent modeling steps. The baseline emulates automata on the CPU (Aho and Corasick, 1975).

Often the input is a collection of documents, each of which should be classified independently. To separate documents, we have the hardware match document boundaries, such as newlines, and output a special value. Since the hardware natively reports matches in order by start position (then by end position), the special value acts as a delimiter between documents that the CPU can detect. This removes the need to reconcile document offsets on the CPU and saves bus bandwidth since the hardware can be configured to not report offsets.

5 Language Model Probability

The task is to compute the language model probability p of some text w. Backoff models (Katz, 1987) memorize probability for seen n–grams and charge a backoff penalty b for unseen n–grams.

$$p(w_n \mid w_1^{n-1}) = \begin{cases} p(w_n \mid w_1^{n-1}) \text{ if } w_1^n \text{ is seen} \\ p(w_n \mid w_2^{n-1}) b(w_1^{n-1}) \text{ o.w.} \end{cases}$$

5.1 Optimizing the Task

The backoff algorithm normally requires storing probability p and backoff b with each seen n–gram. However, Heafield et al. (2012) used telescoping series to prove that probability and backoff can be collapsed into a single function q

$$q(w_n|w_1^{n-1}) = p(w_n|w_1^{n-1}) \frac{\prod_{i=1}^n b(w_i^n)}{\prod_{i=1}^{n-1} b(w_i^{n-1})}$$

This preserves sentence-level probabilities.[1]

Because the hardware lacks user-accessible arithmetic, terms are sent to the CPU. Sending just q for each token instead of p and various backoffs b reduces communication and CPU workload. We also benefit from a simplified query procedure: for each word, match as much context as possible then return the corresponding value q.

5.2 Greedy Matching

Language models are greedy in the sense that, for every word, they match as much leading context as possible. We map this onto greedy regular expressions, which match as much trailing context as possible, by reversing the input and n–grams.[2]

Unlike language identification, we run the hardware in a greedy mode that scans until a match is found, reports the longest such match, and resumes scanning afterwards. The trailing context operator / allows fine-grained control over the offset where scanning resumes. Given two regular expressions r and s, the trailing context expression r/s matches rs as if they were concatenated, but scanning resumes after r. For example, if the language model contains n–gram "This is a", then we create regular expression

```
" a"/" is This "
```

where the quotes ensure that spaces are interpreted literally. Scanning resumes at the space before the next word: " is". Because greedy mode suppresses shorter matches, only the longest n–gram will be reported. The CPU can then sum $\log q$ values associated with each expression without regard to position.

Unknown words are detected by matching a space: `" "`. Vocabulary words will greedily

[1] Technically, q is off by the constant $b(\text{<s>})$ due to conditioning on <s>. We account for this at the end of sentence, re-defining $q(\text{</s>} \mid w_1^{n-1}) \leftarrow q(\text{</s>} \mid w_1^{n-1}) b(\text{<s>})$. Doing so saves one output per sentence.

[2] For exposition, we show words in reverse order. The implementation reverses bytes.

Rule	Value	Purpose
`" a"/" in "`	$q(\text{a} \mid \text{in})$	Normal query
`" "`	$q(\text{<unk>})$	Unknown word
`" in"/" \n"`	$q(\text{in} \mid \text{<s>})$	Sentence begin
`" \n"/" "`	$q(\text{</s>})$	Sentence end
`" \n"/" in "`	$q(\text{</s>} \mid \text{in})$	Sentence end

Table 1: Example regular expressions, including the special rules for the unknown word and sentence boundaries. We rely on the newline `\n` in lieu of sentence boundary tokens <s> and </s>.

Model	Platform	1 core	5 cores
langid	Hardware	160.34	608.41
	C	64.57	279.18
	Java	25.53	102.72
	Python	2.90	12.63
CLD2	C++	12.39	30.15

Table 2: Language identification speed in MB/s.

match their own regular expression, which begins with a space. This space also prevents matching inside an unknown word (e.g. "Ugrasena" should not match "a"). The tokenizer is expected to remove duplicate spaces and add them at line boundaries. Table 1 shows key expressions.

Instead of strings, we can match vocabulary indices. Spaces are unnecessary since indices have fixed length and the unknown word has an index.

6 Experiments

We benchmarked a Tarari T2540 PCI express device from 2011 against several CPU baselines. It has 2 GB of DDR2 RAM and 5 cores. A single-threaded CPU program controls the device and performs arithmetic. The program scaled linearly to control four devices, so it is not a bottleneck. Wall clock time, except loading, is the minimum from three runs on an otherwise-idle machine. Models and input were in RAM before each run.

6.1 Language Identification

The `langid.py` model is 88.6–99.2% accurate (Lui and Baldwin, 2012). We tested the original Python, a Java implementation that "should be faster than anything else out there" (Weiss, 2013), a C implementation (Lui, 2014), and our replica in hardware. We also tested CLD2 (Sites, 2013) written in C++, which has a different model that was less accurate on 4 of 6 languages selected from Europarl (Koehn, 2005). Time includes the costs of feature extraction and modeling.

Lines	Tokens	Ken	DA	1 core	5 cores
100	$2.6 \cdot 10^3$	37.8	40.3	6.6	2.1
1000	$2.2 \cdot 10^4$	42.4	43.6	16.2	10.7
10000	$2.6 \cdot 10^5$	53.9	55.7	46.2	42.0
100000	$2.8 \cdot 10^6$	78.6	85.3	91.3	93.6
305263	$8.6 \cdot 10^6$	92.9	105.6	97.0	91.8

Table 3: Seconds to compute perplexity on strings. The hardware was tested with 1 core and 5 cores.

Table 2 reports speed measured on a 9.6 GB text file created by concatenating the 2013 News Crawl corpora for English, French, German, Hindi, Spanish, and Russian (Bojar et al., 2014). One hardware core is 2.48 times as fast as the fastest CPU program. Using five cores instead of one yielded speed improvements of 3.8x on hardware and 4.3x on a 16-core CPU. The hardware performs decently on this task, likely because the 1 MB binary transition table mostly fits in cache.

6.2 Language Modeling

We benchmarked against the fastest reported language models, DALM's reverse trie (Yasuhara et al., 2013) and KenLM's linear probing (Heafield, 2011). Both use stateful queries. For surface strings, time includes the cost of vocabulary lookup. For vocabulary identifiers, we converted words to bytes then timed custom query programs.

Unpruned models were trained on the English side of the French–English MultiUN corpus (Eisele and Chen, 2010). Perplexity was computed on 2.6 GB of tokenized text from the 2013 English News Crawl (Bojar et al., 2014).

6.2.1 Surface Strings

We tested trigram language models trained on various amounts of data before reaching a software-imposed limit of 4.2 million regular expressions.[3] Figure 1 and Table 3 show total query time as a function of training data size while Figure 2 shows model size. DALM model size includes the entire directory.

Cache effects are evident: the hardware binary format is much larger because it stores a generic table. Queries are fast for tiny models but become slower than the CPU. Multiple cores do not help for larger models because they share the cache and memory bus. Since the hardware operates at the byte level and there is an average of 5.34 bytes

[3] Intel is working to remove this restriction.

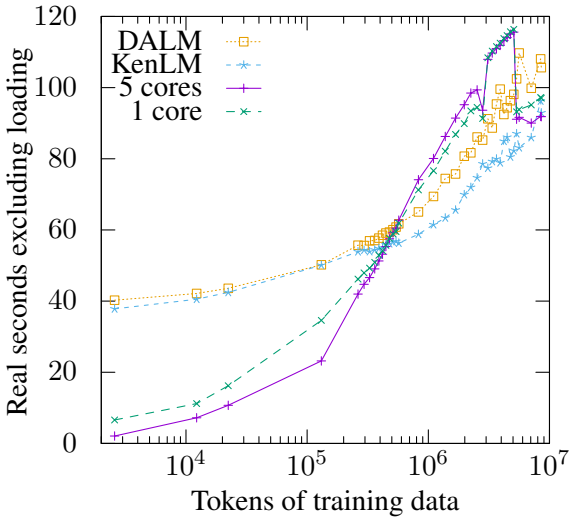

Figure 1: Time to compute perplexity on strings.

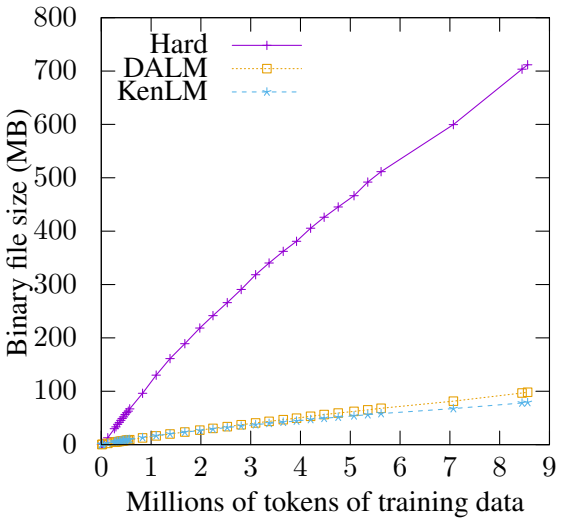

Figure 2: Size of the models on strings.

per word, random memory accesses happen more often than in CPU-based models that operate on words. We then set out to determine if the hardware runs faster when each word is a byte.

6.2.2 Vocabulary Indices

Class-based language models are often used alongside lexical language models to form generalizations. We tested a 5–gram language model over CoNLL part-of-speech tags from MITIE (King, 2014). There are fewer than 256 unique tags, fitting into a byte per word. We also created special KenLM and DALM query programs that read byte-encoded input. Figure 3 and Table 4 show total time while model sizes are shown in Figure 4. Performance plateaus for very small models, which is more clearly shown by plotting speed in Figure 5.

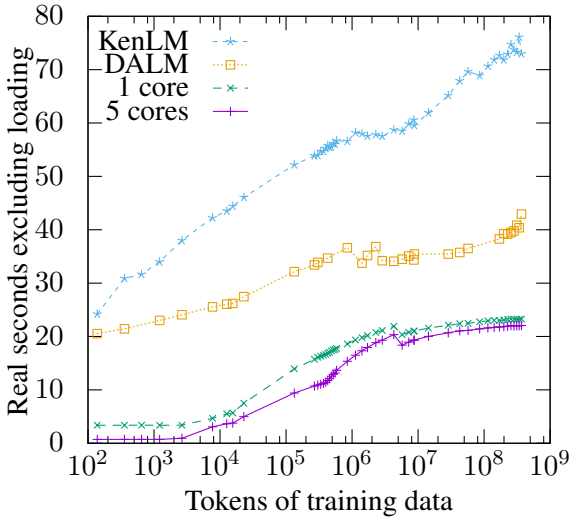

Figure 3: Time to compute perplexity on bytes.

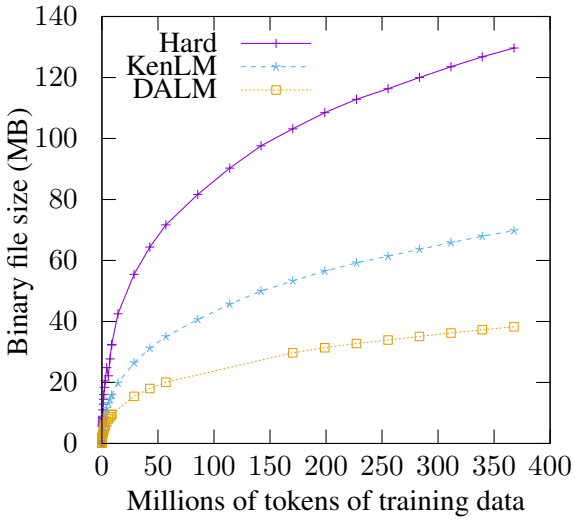

Figure 4: Size of the models on bytes.

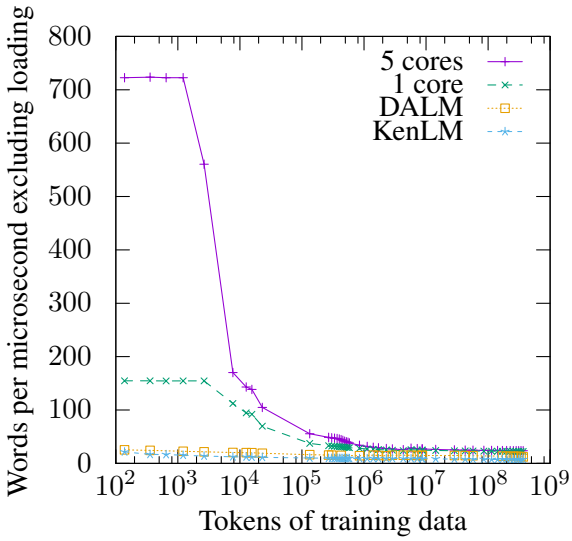

Figure 5: Speed, in words per microsecond, to compute perplexity on bytes.

Lines	Tokens	Ken	DA	1 core	5 cores
100	$2.6 \cdot 10^3$	38.0	24.1	3.4	0.9
1000	$2.3 \cdot 10^4$	46.1	27.5	7.5	5.0
10000	$2.7 \cdot 10^5$	53.9	33.4	15.7	10.7
100000	$2.9 \cdot 10^6$	57.5	34.2	21.1	19.3
1000000	$2.9 \cdot 10^7$	65.2	35.4	22.1	20.7
13000000	$3.7 \cdot 10^8$	73.0	42.9	23.3	22.0

Table 4: Seconds to compute perplexity on bytes. The hardware was tested with 1 core and 5 cores.

The hardware is faster for all training data sizes we tested. For tiny models, one core is initially 6 times as fast one CPU core while larger models are 1.8 times as fast as the CPU. For small models, the hardware appears to hitting another limit, perhaps the speed at which a core can output matches. This is not a CPU or PCI bus limitation because five cores are faster than one core, by a factor of 4.67.

Model growth is sublinear because novel POS n–grams are limited. The hardware binary image is 3.4 times as large as DALM, compared with 7.2 times as large for the lexical model. We attribute this to denser transition tables that result from model saturation.

Acknowledgements

We thank Intel and Xanadata for numerous consultations and for providing access to a machine with four devices. Intel markets the hardware as a regular expression processor, not a deterministic pushdown automaton.

7 Conclusion

Language identification and language modeling entail scanning that can be offloaded to regular expression hardware. The hardware works best for small models, such as those used in language identification. Like CPUs, random memory accesses are slow. We believe it will be useful for web-scale extraction problems, where language identification and coarse language modeling are used to filter large amounts of data. We plan to investigate a new hardware version that Intel is preparing.

References

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. Openfst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata*, pages 11–23. Springer.

Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. 2014. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 12–58, Baltimore, Maryland, USA, June. Association for Computational Linguistics.

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

Cristian Ciressan, Eduardo Sanchez, Martin Rajman, and Jean-Cedric Chappelier. 2000. An fpga-based coprocessor for the parsing of context-free grammars. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pages 236–236. IEEE Computer Society.

Andreas Eisele and Yu Chen. 2010. MultiUN: A multilingual corpus from United Nation documents. In Daniel Tapias, Mike Rosner, Stelios Piperidis, Jan Odjik, Joseph Mariani, Bente Maegaard, Khalid Choukri, and Nicoletta Calzolari, editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation*, pages 2868–2872. European Language Resources Association (ELRA), 5.

Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, Brisbane, Australia.

David Hall, Taylor Berg-Kirkpatrick, John Canny, and Dan Klein. 2014. Sparser, better, faster GPU parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 208–217, June.

Hua He, Jimmy Lin, and Adam Lopez. 2015. Gappy pattern matching on GPUs for on-demand extraction of hierarchical translation grammars. *Transactions of the Association for Computational Linguistics*, 3:87–100.

Kenneth Heafield, Philipp Koehn, and Alon Lavie. 2012. Language model rest costs and space-efficient storage. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Jeju Island, Korea.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Edinburgh, UK, July. Association for Computational Linguistics.

Mark Johnson. 2011. Parsing in parallel on multiple cores and GPUs. In *Proceedings of the Australasian Language Technology Association Workshop 2011*, pages 29–37, December.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(3):400–401, March.

Davis E. King. 2014. MITIE: MIT information extraction, January. `https://github.com/mit-nlp/MITIE`.

Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of MT Summit*.

Jan Korenek. 2010. Fast regular expression matching using FPGA. *Information Sciences and Technologies Bulletin of the ACM Slovakia*, 2(2):103–111.

Jooyoung Lee, Sungkyong Un, and Dowon Hong. 2008. High-speed search using Tarari content processor in digital forensics. *Digital Investigation*, 5:S91–S95.

Michael E Lesk and Eric Schmidt. 1975. Lex: A lexical analyzer generator, July.

Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. 2006. Optimization of regular expression pattern matching circuits on FPGA. In *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, volume 2, pages 1–6. IEEE.

Marco Lui and Timothy Baldwin. 2011. Cross-domain feature selection for language identification. In *Proceedings of the 5th International Joint Conference on Natural Language Processing*, pages 553–561, Chiang Mai, Thailand, November.

Marco Lui and Timothy Baldwin. 2012. `langid.py`: An off-the-shelf language identification tool. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 25–30, Jeju, Republic of Korea, July.

Marco Lui. 2014. Pure C natural language identifier with support for 97 languages. `https://github.com/saffsd/langid.c`.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2000. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32.

Kyoung-Su Oh and Keechul Jung. 2004. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314.

Isaac Rudomín, Erik Millán, and Benjamín Hernández. 2005. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory*, 13(8):741–751.

Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE.

Dick Sites. 2013. Compact language detection 2. `https://code.google.com/p/cld2/`.

Jason R. Smith, Herve Saint-Amand, Magdalena Plamada, Philipp Koehn, Chris Callison-Burch, and Adam Lopez. 2013. Dirt cheap web-scale parallel text from the common crawl. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria, August.

Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *Proceedings of the Seventh International Conference on Spoken Language Processing*, pages 901–904.

Dawid Weiss. 2013. Java port of langid.py (language identifier). `https://github.com/carrotsearch/langid-java`.

Makoto Yasuhara, Toru Tanaka, Jun-ya Norimatsu, and Mikio Yamamoto. 2013. An efficient language model using double-array structures. In *Proceedings of EMNLP*, pages 222–232, October.

Youngmin Yi, Chao-Yue Lai, Slav Petrov, and Kurt Keutzer. 2011. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 175–185. Association for Computational Linguistics.