Gappy Pattern Matching on GPUs for On-Demand Extraction of Hierarchical Translation Grammars

Hua He
Dept. of Computer Science
University of Maryland
College Park, Maryland
huah@cs.umd.edu

Jimmy Lin
The iSchool and UMIACS
University of Maryland
College Park, Maryland
jimmylin@umd.edu

Adam Lopez
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
alopez@inf.ed.ac.uk

Abstract

Grammars for machine translation can be materialized on demand by finding source phrases in an indexed parallel corpus and extracting their translations. This approach is limited in practical applications by the computational expense of online lookup and extraction. For *phrase-based* models, recent work has shown that on-demand grammar extraction can be greatly accelerated by parallelization on general purpose graphics processing units (GPUs), but these algorithms do not work for *hierarchical* models, which require matching patterns that contain gaps. We address this limitation by presenting a novel GPU algorithm for on-demand hierarchical grammar extraction that is at least an order of magnitude faster than a comparable CPU algorithm when processing large batches of sentences. In terms of end-to-end translation, with decoding on the CPU, we increase throughput by roughly two thirds on a standard MT evaluation dataset. The GPU necessary to achieve these improvements increases the cost of a server by about a third. We believe that GPU-based extraction of hierarchical grammars is an attractive proposition, particularly for MT applications that demand high throughput.

1 Introduction

Most machine translation systems extract a large, fixed translation model from parallel text that is accessed from memory or disk. An alternative is to store the indexed parallel text in memory and extract translation units on demand only when they are needed to decode new input. This architecture has several advantages: It requires only a few gigabytes to represent a model that would otherwise require a terabyte (Lopez, 2008b). It can adapt incrementally to new training data (Levenberg et al., 2010), making it useful for interactive translation (González-Rubio et al., 2012). It supports rule extraction that is sensitive to the input sentence, enabling leave-one-out training (Simianer et al., 2012) and the use of sentence similarity features (Philips, 2012).

On-demand extraction can be slow, but for phrase-based models, massive parallelization on general purpose graphics processing units (GPUs) can dramatically accelerate performance. He et al. (2013) demonstrated orders of magnitude speedup in exact pattern matching with suffix arrays, the algorithms at the heart of on-demand extraction (Callison-Burch et al., 2005; Zhang and Vogel, 2005). However, some popular translation models use "gappy" phrases (Chiang, 2007; Simard et al., 2005; Galley and Manning, 2010), and the GPU algorithm of He et al. does not work for these models since it is limited to contiguous phrases. Instead, we need pattern matching and phrase extraction that is able to handle variable-length gaps (Lopez, 2007).

This paper presents a novel GPU algorithm for on-demand extraction of hierarchical translation models based on matching and extracting gappy phrases. Our experiments examine both grammar extraction and end-to-end translation, comparing quality, speed, and memory use. We compare against the GPU system for phrase-based translation by He et al. (2013) and cdec, a state-of-the-art

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

Transactions of the Association for Computational Linguistics, vol. 3, pp. 87–100, 2015. Action Editor: David Chiang.
Submission batch: 6/2014; Revision batch 12/2014; Published 2/2015. ©2015 Association for Computational Linguistics.

CPU system for hierarchical translation (Dyer et al., 2010). Our system outperforms the former on translation quality by 2.3 BLEU (replicating previously-known results) and outperforms the latter on speed, improving grammar extraction throughput by at least an order of magnitude on large batches of sentences while maintaining the same level of translation quality. Our contribution is to show, complete with an open-source implementation, how GPUs can vastly increase the speed of hierarchical grammar extraction, particularly for high-throughput MT applications.

2 Algorithms

GPU architectures, which are optimized for massively parallel operations on relatively small streams of data, strongly influence the design of our algorithms, so we first briefly review some key properties. Our NVIDIA Tesla K20c GPU (Kepler Generation) provides 2496 thread processors (CUDA cores), and computation proceeds concurrently in groups of 32 threads called *warps*. Each thread in a warp carries out identical instructions in lockstep. When a branching instruction occurs, only threads meeting the branch condition execute, while the rest idle—this is called *warp divergence* and is a source of poor performance. Consequently, GPUs are poor at "irregular" computations that involve conditionals, pointer manipulation, and complex execution sequences.

Our pattern matching algorithm is organized around two general design principles: brute force scans and fine-grained parallelism. Brute force array scans avoid warp divergence since they access data in regular patterns. Rather than parallelize larger algorithms that use these scans as subroutines, we parallelize the scans themselves in a fine-grained manner to obtain high throughput.

The relatively small size of the GPU memory also affects design decisions. Data transfer between the GPU and the CPU has high latency, so we want to avoid shuffling data as much as possible. To accomplish this, we must fit all our data structures into the 5 GB memory available on our particular GPU. As we will show, this requires some tradeoffs in addition to careful design of algorithms and associated data structures.

2.1 Translation by Pattern Matching

Lopez (2008b) provides a recipe for "translation by pattern matching" that we use as a guide for the remainder of this paper (Algorithm 1).

Algorithm 1 Translation by pattern matching

```
1: for each input sentence do
2:   for each phrase in the sentence do
3:     Find its occurrences in the source text
4:     for each occurrence do
5:       Extract any aligned target phrase
6:     for each extracted phrase pair do
7:       Compute feature values
8:   Decode as usual using the scored rules
```

We encounter a computational bottleneck in lines 2–7, since there are many query phrases, matching occurrences, and extracted phrase pairs to process. Below, we tackle each challenge in turn.

To make our discussion concrete, we will use a toy English-Spanish translation example. At line 3 we search for phrases in the source side of a parallel text. Suppose that it contains two sentences: *it makes him and it mars him*, and *it sets him on and it takes him off*. We map each unique word to an integer id, and call the resulting array of integers that encodes the source text under this transformation the text T. Let $|T|$ denote the total number of tokens, $T[i]$ denote its ith element, and $T[i]...T[j]$ denote the substring starting with its ith element and ending with its jth element. Since T encodes multiple sentences, we use special tokens to denote the end of a sentence and the end of the corpus. In our example we use # and $, respectively, as shown in Figure 1. Now suppose we want to translate the sentence *it persuades him and it disheartens him*. We encode it under the same mapping as T and call the resulting array the query Q.

Our goal is to materialize all of the hierarchical phrases that could be used to translate Q based on training data T. Our algorithm breaks this process into a total of 14 distinct passes, each performing a single type of computation in parallel. Five of these passes are based on the algorithm described by He et al. (2013), and we review them for completeness. The nine new passes are identified as such.

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

i	$T[i]$	$S_T[i]$	suffix $S_T[i]$: $T[S_T[i]]...T[\|T\|]$
1	#	5	and it mars him # it sets him on and it takes him off # $
2	it	14	and it takes him off # $
3	makes	4	him and it mars him # it sets him on and it takes him off # $
4	him	17	him off # $
5	and	12	him on and it takes him off # $
6	it	8	him # it sets him on and it takes him off # $
7	mars	2	it makes him and it mars him # it sets him on and it takes him off # $
8	him	6	it mars him # it sets him on and it takes him off # $
9	#	10	it sets him on and it takes him off # $
10	it	15	it takes him off # $
11	sets	3	makes him and it mars him # it sets him on and it takes him off # $
12	him	7	mars him # it sets him on and it takes him off # $
13	on	18	off # $
14	and	13	on and it takes him off # $
15	it	11	sets him on and it takes him off # $
16	takes	16	takes him off # $
17	him	1	# it makes him and it mars him # it sets him on and it takes him off # $
18	off	9	# it sets him on and it takes him off # $
19	#	19	# $
20	$	20	$

Figure 1: Example text T (showing words rather than than their integer encodings for clarity) and suffix array S_T with corresponding suffixes.

2.2 Finding Every Phrase

Line 3 of Algorithm 1 searches T for all occurrences of all phrases in Q. We call each phrase a pattern, and our goal is to find all phrases of T that match each pattern, i.e., the problem of pattern matching. Our translation model permits phrases with at most two gaps, so Q is a source of $\mathcal{O}(|Q|^6)$ patterns, since there are up to six possible subphrase boundaries.

Passes 1-2: Finding contiguous patterns

To find contiguous phrases (patterns without gaps), we use the algorithm of He et al. (2013). It requires a suffix array (Manber and Myers, 1990) computed from T. The ith suffix of a 1-indexed text T is the substring $T[i]...T[|T|]$ starting at position i and continuing to the end of T. The suffix array S_T is a permutation of the integers $1, ..., |T|$ ordered by a lexicographic sort of the corresponding suffixes (Figure 1). Given S_T, finding a pattern P is simply a matter of binary search for the pair of integers (ℓ, h) such that for all i from ℓ to h, P is a prefix of the $S_T[i]$th suffix of T. Thus, each integer $S_T[i]$ identifies a unique match of P. In our example, the pattern *it* returns $(7, 10)$, corresponding to matches at positions 2, 6, 10, and 15; while *him and it* returns $(3, 3)$, corresponding to a match at position 4. A longest common prefix (LCP) array enables us to find h or ℓ in $\mathcal{O}(|Q| + \log |T|)$ comparisons (Manber and Myers, 1990).

Every substring of Q is a contiguous pattern, but if we searched T for all of them, most searches would fail, wasting computation. Instead, He et al. (2013) use two passes. The first computes, concurrently for every position i in $1, ..., |Q|$, the endpoint j of the longest substring $Q[i]...Q[j]$ that appears in T. It also computes the suffix array range of the one-word substring $Q[i]$. Taking this range as input, for all k from i to j the second pass concurrently queries T for pattern $Q[i]...Q[k]$. This pass uses two concurrent threads per pattern—one to find the lowest index of the suffix array range, and one to find the highest index.

Passes 3-4: Finding one-gap patterns (New)

Passes 1 and 2 find contiguous phrases, but we must

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

also find phrases that contain gaps. We use the special symbol $\star$ to denote a variable-length gap. The set of one-gap patterns in Q thus consists of $Q[i]...Q[j] \star Q[i']...Q[j']$ for all i, j, i', and j' such that $i \leq j < i' - 1$ and $i' \leq j'$. When the position in Q is not important we use strings u, v, and w to denote contiguous patterns; for example, $u \star v$ denotes an arbitrary one-gap pattern. We call the contiguous strings u and v of $u \star v$ its subpatterns, e.g., *it* $\star$ *him* is a pattern with subpatterns *it* and *him*.

When we search for a gappy pattern, $\star$ can match any non-empty substring of T that does not contain $ or #. Such a match may not be uniquely identified by the index of its first word, so we specify it with a tuple of indices, one for the match of each subpattern. Pattern *it* $\star$ *him* has six matches in T: $(2, 4)$, $(2, 8)$, $(6, 8)$, $(10, 12)$, $(10, 17)$, and $(15, 17)$. Passes 3 and 4 search T for all one-gap patterns using the novel GPU algorithm described below.

A pattern $u \star v$ cannot match in T unless both u and v match in T, so we use the output of pass 1, which returns all (i, j) pairs such that $Q[i]...Q[j]$ matches in T. Concurrently for every such i and j, pass 3 enumerates all i' and j' such that $j < i' - 1$ and $Q[i']...Q[j']$ matches in T, returning each pattern $Q[i]...Q[j] \star Q[i']...Q[j']$. Pass 3 then sorts and deduplicates the combined results of all threads to obtain a set of unique patterns. These operations are carried out on the GPU using the algorithms of Hoberock and Bell (2010).

Pass 4 searches for matches of each pattern identified by pass 3. We first illustrate with *it* $\star$ *him*. Pass 2 associates *it* with suffix array range $(7, 10)$. A linear scan of S_T in this range reveals that *it* matches at positions 2, 6, 10, and 15 in T. Likewise, *him* maps to range $(3, 6)$ of S_T and matches at 4, 17, 12, and 8 in T. Concurrently for each match of the less frequent subpattern, we scan T to find matches of the other subpattern until reaching a sentence boundary or the maximum phrase length, an idea we borrow from the CPU implementation of Baltescu and Blunsom (2014). In our example, both *it* and *him* occur an equal number of times, so we arbitrarily choose one—suppose we choose *it*. We assign each of positions 2, 6, 10, and 15 to a separate thread. The thread assigned position 2 scans T for matches of *him* until the end of sentence at position 9, finding matches $(2, 4)$ and $(2, 8)$.

As a second example, consider *it* $\star$ *and*. In this case, *it* has four matches, but *and* only two. So, we need only two threads, each scanning backwards from matches of *and*. Since most patterns are infrequent, allocating threads this way minimizes work. However, very large corpora contain one-gap patterns for which both subpatterns are frequent. We simply precompute all matches for these patterns and retrieve them at runtime, as in Lopez (2007). This precomputation is performed once given T and therefore it is a one-time cost.

Materializing every match of $u \star v$ would consume substantial memory, so we only emit those for which a translation of the substring matching $\star$ is extractable using the check in §2.3. The success of this check is a prerequisite for extracting the translation of $u \star v$ or any pattern containing it, so pruning in this way conserves GPU memory without affecting the final grammar.

Passes 5-7: Finding two-gap patterns (New) We next find all patterns with two gaps of the form $u \star v \star w$. The search is similar to passes 3 and 4. In pass 5, concurrently for every pattern $u \star v$ matched in pass 4, we enumerate the pattern $u \star v \star w$ for every w such that $u \star v \star w$ is a pattern in Q and w matched in pass 1. In pass 6, concurrently for every match (i, j) of $u \star v$ for every $u \star v \star w$ enumerated in pass 5, we scan T from position $j + |v| + 1$ for matches of w until we reach the end of sentence. As with the one-gap patterns, we apply the extraction check on the second $\star$ of the two-gap patterns $u \star v \star w$ to avoid needlessly materializing matches that will not yield translations.

2.3 Extracting Every Target Phrase

In line 5 of Algorithm 1 we must extract the aligned translation of every match of every pattern found in T. Efficiency is crucial since some patterns may occur hundreds of thousands of times.

We extract translations from word alignments using the consistency check of Och et al. (1999). A pair of substrings is consistent only if no word in either substring is aligned to any word outside the pair. For example, in Figure 2 the pair (*it sets him on, los excita*) is consistent. The pair (*him on and, los excita y*) is not, because *excita* also aligns to the words *it sets*. Only consistent pairs can be

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

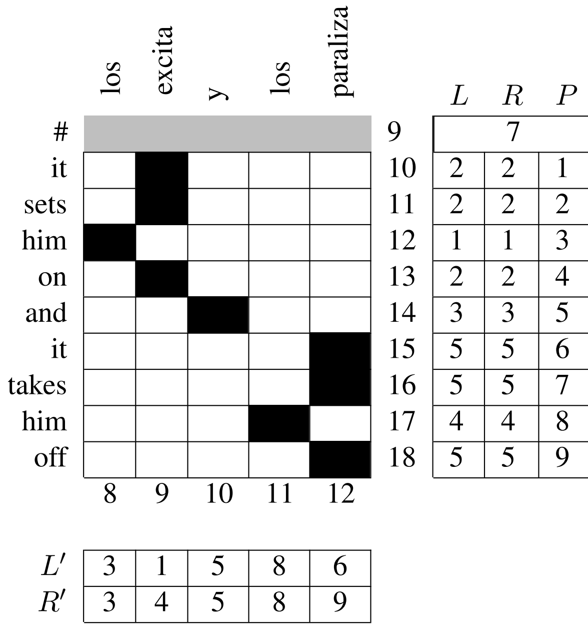

Figure 2: An example alignment and the corresponding L, R, P, L' and R' arrays.

translations of each other in our model.

Given a specific source substring, our algorithm asks: is it part of a consistent pair? To answer this question, we first compute the minimum target substring to which all words in the substring align. We then compute the minimum substring to which all words of this candidate translation align. If this substring matches the input, the candidate translation is returned; otherwise, extraction fails. For example, *it sets him on* in range $(10, 13)$ aligns to *los excita* in range $(8, 9)$, which aligns back to $(10, 13)$. So this is a consistent pair. However, *him on and* in range $(12, 14)$ aligns to *los excita y* in range $(8, 10)$, which aligns back to *it sets him on and* at $(10, 14)$. So *him on and* is not part of a consistent pair and has no extractable translation.

To extract gappy translation units, we subtract consistent pairs from other consistent pairs (Chiang, 2007). For example, (*him, los*) is a consistent pair. Subtracting it from (*it sets him on, los excita*) yields the translation unit (*it sets* ⋆ *on,* ⋆ *excita*).

Our basic building block is the EXTRACT function of He et al. (2013), which performs the above check using byte arrays denoted L, R, P, L', and R' (Figure 2) to identify extractable target phrases in target text T'. When $T[i]$ is a word, $L[i]$ and $R[i]$ store the sentence-relative positions of the leftmost and rightmost words it aligns to in T', and $P[i]$ stores $T[i]$'s sentence-relative position. When $T[i]$ is a sentence boundary, the concatenation of bytes $L[i]$, $R[i]$, and $P[i]$, denoted $LRP[i]$, stores the position of the corresponding sentence in T'. Bytes $L'[i']$ and $R'[i']$ store the sentence-relative positions of the leftmost and rightmost words $T'[i']$ aligns to.

We first calculate the start position p of the source sentence containing $T[i]...T[j]$, and the start position p' of the corresponding target sentence:

$$p = i - P[i]$$
$$p' = LRP[p]$$

We then find target indices i' and j' for the candidate translation $T'[i']...T'[j']$:

$$i' = p' + \min_{k \in i,...,j} L[k]$$
$$j' = p' + \max_{k \in i,...,j} R[k]$$

We can similarly find the translation $T[i'']...T[j'']$ of $T'[i']...T'[j']$:

$$i'' = p + \min_{k' \in i',...,j'} L'[k']$$
$$j'' = p + \max_{k' \in i',...,j'} R'[k']$$

If $i = i''$ and $j = j''$, EXTRACT(i, j) returns (i', j'), the position of $T[i]...T[j]$'s translation. Otherwise the function signals that there is no extractable translation. Given this function, extraction proceeds in three passes.

Pass 8: Extracting contiguous patterns

Each match of pattern u is assigned to a concurrent thread. The thread receiving the match at position i returns the pair consisting of u and its translation according to EXTRACT$(i, i + |u| - 1)$, if any. It also returns translations for patterns in which u is the only contiguous subpattern: $\star\, u$, $u\, \star$, and $\star\, u\, \star$. We extract translations for these patterns even if u has no translation itself. To see why, suppose that we reverse the translation direction of our example. In Figure 2, *excita* is not part of a consistent pair, but both ⋆ *excita* and ⋆ *excita* ⋆ are.

Consider $\star\, u$. Since ⋆ matches any substring in T without boundary symbols, the leftmost position of

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

$\star\ u$ is not fixed. So, we seek the smallest match with an extractable translation, returning its translation with the following algorithm.

```
1:  k ← i
2:  while T[k − 1] ≠ # do
3:      k ← k − 1
4:      if EXTRACT(k, i + |u| − 1) succeeds then
5:          (i′, j′) ← EXTRACT(k, i + |u| − 1)
6:          if EXTRACT(k, i − 1) succeeds then
7:              (p′, q′) ← EXTRACT(k, i − 1)
8:              return T′[i′]...T′[p′] ⋆ T′[q′]...T′[j′]
9:  if T[k − 1] = # then
10:     return failure
```

The case of $u\ \star$ is symmetric.

We extend this algorithm to handle $\star\ u\ \star$. The extension considers increasingly distant pairs (k, ℓ) for which $\star\ u\ \star$ matches $T[k]...T[\ell]$, until it either finds an extractable translation, or it encounters both sentence boundaries and fails.

Pass 9: Extracting one-gap patterns (New)
In this pass, each match of pattern $u\ \star\ v$ is assigned to a thread. The thread receiving match (i, j) attempts to assign i', j', p' and q' as follows.

$$(i', j') = \text{EXTRACT}(i, j + |v| - 1)$$
$$(p', q') = \text{EXTRACT}(i + |u|, j - 1)$$

If both calls succeed, we subtract to obtain the translation $T'[i']...T'[p'] \star T'[q']...T'[j']$. We extract translations for $\star\ u\ \star\ v$ and $u\ \star\ v\ \star$, using the same algorithm as in pass 7.

Pass 10: Extracting two-gap patterns (New)
In this pass, we extract a translation for each match of $u\ \star\ v\ \star\ w$ concurrently, using a straightforward extension of the subtraction algorithm in pass 8. Since we only need patterns with up to two gaps, we do not consider other patterns.

To optimize GPU performance, for the passes above, we assign all matches of a gappy pattern to the same thread block. This allows threads in the same thread block to share the same data during initialization, therefore improving memory access coherence.

2.4 Computing Every Feature

In line 7 of Algorithm 1 we compute features of every extracted phrase pair. We use α and β to denote arbitrary strings of words and $\star$ symbols, and our input is a multiset of (α, β) pairs collected by passes 7-9, which we denote by Π. We compute the following features for each unique (α, β) pair.

Log-count features. We need two aggregate statistics: The count of (α, β) in Π, and the count of all pairs in Π for which α is the source pattern. We then compute the two features as $\log(1 + \text{count}(\alpha, \beta))$ and $\log(1 + \text{count}(\alpha))$.

Translation log-probability. Given the aggregate counts above, this feature is $\log \frac{\text{count}(\alpha,\beta)}{\text{count}(\alpha)}$.

Singleton indicators. We compute two features, to indicate whether (α, β) occurs only once, i.e., $\text{count}(\alpha, \beta) = 1$, and whether α occurs only once, i.e., $\text{count}(\alpha) = 1$.

Lexical weight. Consider word pairs a, b with $a \in \alpha$, $b \in \beta$, and neither a nor b are $\star$. Given a global word translation probability table $p(a|b)$, which is externally computed from the word alignments directly, the feature is $\sum_{a\in\alpha} \max_{b\in\beta} \log p(a|b)$.

Since Π is the result of parallel computation, we must sort it. We can then compute aggregate statistics by keeping running totals in a scan of the sorted multiset. With many instantiated patterns, we would quickly exhaust GPU memory, so this sort is performed on the CPU. We compute the log-count features, translation log-probability, and singleton indicators this way. However, the lexical weight feature is a function only of the aligned translation pair itself and corresponding word-level translation possibilities calculated externally from word alignment. Thus, the computation of this feature can be parallelized on the GPU. So, we have multiple feature extraction passes based on the number of gaps:

Pass 11 (CPU): One-gap features (New).
Pass 12 (CPU): Two-gap features (New).
Pass 13 (CPU): Contiguous features.
Pass 14 (GPU): Lexical weight feature (New).

2.5 Sampling

In serial implementations of on-demand extraction, very frequent patterns are a major computational bottleneck (Callison-Burch et al., 2005; Lopez, 2008b). Thus, for patterns occurring more than n times, for some fixed n (typically

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

between 100 and 1000), these implementations deterministically sample n matches, and only extract translations of these matches. To compare with these implementations, we also implement an optional sampling step. Though we use the same sampling rate, the samples themselves are not the same, since our extraction checks in passes 4 and 6 alter the set of matches that are actually enumerated, thus sampled from. The CPU algorithms do not use this check.

3 Experimental Setup

We tested our algorithms in an end-to-end Chinese-English translation task using data conditions similar to those of Lopez (2008b) and He et al. (2013). Our implementation of hierarchical grammar extraction on the GPU, as detailed in the previous section, is written in C, using CUDA library v5.5 and GCC v4.8, compiled with the -O3 optimization flag. Our code is open source and available for researchers to download and try out.[1]

Hardware. We used NVIDIA's Tesla K20c GPU (Kepler Generation), which has 2496 CUDA cores and 5 GB memory, with a peak memory bandwidth of 208 GB/s. The server hosting the GPU has two Intel Xeon E5-2690 CPUs, each with eight cores at 2.90 GHz (a total of 16 physical cores; 32 logical cores with hyperthreading). Both were released in 2012 and represent comparable generation hardware technology. All GPU and CPU experiments were conducted on the same machine, which runs Red Hat Enterprise Linux (RHEL) 6.

Training Data. We used two training sets: The first consists of news articles from the Xinhua Agency, with 27 million words of Chinese (around one million sentences). The second adds parallel text from the United Nations, with 81 million words of Chinese (around four million sentences).

Test Data. For performance evaluations, we ran tests on sentence batches of varying sizes: 100, 500, 1k, 2k, 4k, 6k, 8k, 16k and 32k. These sentences are drawn from the NIST 2002–2008 MT evaluations (on average 27 words each) and then the Chinese side of the Hong Kong Parallel Text (LDC2004T08) when the NIST data are smaller than the target batch size. Large batch sizes are necessary to saturate the processing power of the GPU. The size of the complete batch of 32k test sentences is 4892 KB.

Baselines. We compared our GPU implementation for on-demand extraction of hierarchical grammars against the corresponding CPU implementation (Lopez, 2008a) found in pycdec (Chahuneau et al., 2012), an extension of cdec (Dyer et al., 2010).[2] We also compared our GPU algorithms against Moses (Koehn et al., 2007), representing a standard phrase-based SMT baseline. Phrase tables generated by Moses are essentially the same as the GPU implementation of on-demand extraction for phrase-based translation by He et al. (2013).

4 Results

4.1 Translation quality

We first verified that our GPU implementation achieves the same translation quality as the corresponding CPU baseline. This is accomplished by comparing system output against the baseline systems, training on Xinhua, tuning on NIST03, and testing on NIST05. In all cases, we used MIRA (Chiang, 2012) to tune parameters. We ran experiments three times and report the average as recommended by Clark et al. (2011). Hierarchical grammars were extracted with sampling at a rate of 300; we also bound source patterns at a length of 5 and matches at a length of 15. For Moses we used default parameters.

Our BLEU scores, shown in Table 1, replicate well-known results where hierarchical models outperform pure phrase-based models on this task. The difference in quality is partly because the phrase-based baseline system does not use lexicalized reordering, which provides similar improvements to hierarchical translation (Lopez, 2008b). Such lexicalized reordering models cannot be produced by the GPU-based system of He et al. (2013). This establishes a clear translation quality improvement between our work and that of He et al. (2013).

[1] http://hohocode.github.io/cgx/

[2] The Chahuneau et al. (2012) implementation is in Cython, a language for building Python applications with performance-critical components in C. All of the pattern matching code that we instrumented for these experiments is compiled to C/C++. The implementation is a port of the original code written by Lopez (2008a) in Pyrex, a precursor to Cython. Much of the code is unchanged.

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

System	BLEU
Moses phrase-based baseline	31.11
Hierarchical with online CPU extraction	33.37
Hierarchical with online GPU extraction	33.46

Table 1: Comparison of translation quality. The hierarchical system is cdec. Online CPU extraction is the baseline, part of the standard cdec package. Online GPU extraction is this work.

threads	+Sampling		−Sampling	
	X	X+U	X	X+U
1	12.7	4.8	0.32	0.05
16	190.7	65.3	4.81	0.70
32	248.1	76.0	6.23	0.89

Table 2: CPU extraction performance (throughput in words/second) using different numbers of threads under different data conditions (X: Xinhua, X+U: Xinhua+UN), with and without sampling.

We see that the BLEU score obtained by our GPU implementation of hierarchical grammar extraction is nearly identical to cdec's, evidence that our implementation is correct. The minor differences in score are due to non-determinism in tuning and the difference in sampling algorithms (§2.5).

4.2 Extraction Speed

Next, we focus on the performance of the hierarchical grammar extraction component, comparing the CPU and GPU implementations. For both implementations, our timings include preparation of queries, pattern matching, extraction, and feature computation. For the GPU implementation, we include the time required to move data to and from the GPU. We do not include time for construction of static data structures (suffix arrays and indexes) and initial loading of the parallel corpus with alignment data, as those represent one-time costs. Note that the CPU implementation includes indexes for frequent patterns in the form $u \star v$ and $u \star v \star w$, while our GPU implementation indexes only the former.

We compared performance varying the number of queries, and following Lopez (2008a), we compared sampling at a rate of 300 against runs without sampling. Our primary evaluation metric is throughput: the average number of processed words per second (i.e., batch size in words divided by total time).

We first establish throughput baselines on the CPU, shown in Table 2. Experiments used different numbers of threads under different data conditions (Xinhua or Xinhua + UN), with and without sampling. Our server has a total of 16 physical cores, but supports 32 logical cores via hyperthreading. We obtained the CPU sampling results by running cdec over 16k query sentences. For the non-sampling runs, since the throughput is so low, we measured performance over 2.6k sentences for Xinhua and 500 sentences for Xinhua + UN. We see that throughput scaling is slightly less than linear: with sampling, using 16 threads increases throughput by 15× on the Xinhua data (compared to a single thread) and 13.6× on Xinhua + UN data. Going from 16 to 32 threads further increase throughput by 15%-30% and saturates the processing capacity of our server. The 32 thread condition provides a fair baseline for comparing the performance of the GPU implementation.

Table 3 shows GPU hierarchical grammar extraction performance in terms of throughput (words/second); these results are averaged over three trials. We varied the number of query sentences, and in each case, also report the speedup with respect to the CPU condition with 32 threads. GPU throughput increases with larger batch sizes because we are increasingly able to saturate the GPU and take full advantage of the massive parallelism it offers. We do not observe this effect on the CPU since we saturate the processors early.

With a batch size of 100 sentences, the GPU is slightly slower than the 32-thread CPU implementation on Xinhua and faster on Xinhua + UN, both with sampling. Without sampling, the GPU is already an order of magnitude faster at a batch size of 100 sentences. At a batch size of 500 sentences, the GPU implementation is substantially faster than the 32-thread CPU version across all conditions. With the largest batch size in our experiments of 32k sentences, the GPU is over an order of magnitude faster than the fully-saturated CPU with sampling, and over two orders of magnitude faster without sampling. Although previous work does not show decreased translation quality due

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

	Batch Size	Number of Sentences Number of Tokens	100 2.8k	500 14.5k	1k 28.8k	2k 57.9k	4k 117.9k	6k 161.9k	8k 214.2k	16k 436.5k	32k 893.9k
+Sampling	Xinhua	Throughput (words/s)	236	667	914	1356	1613	1794	2001	2793	3998
		Speedup	1.0×	2.7×	3.7×	5.5×	6.5×	7.2×	8.1×	11.3×	16.1×
	Xinhua+UN	Throughput (words/s)	106	223	287	400	454	514	571	709	1016
		Speedup	1.4×	2.9×	3.8×	5.3×	6.0×	6.8×	7.5×	9.3×	13.4×
−Sampling	Xinhua	Throughput (words/s)	84	280	405	690	929	1200	1414	2135	3240
		Speedup	13×	45×	65×	111×	149×	193×	227×	343×	520×
	Xinhua+UN	Throughput (words/s)	19	62	99	172	248	304	357	509	793
		Speedup	22×	69×	112×	193×	279×	342×	401×	572×	891×

Table 3: GPU grammar extraction throughput (words/second) under different batch sizes, data conditions, with and without sampling. Speedup is computed with respect to the CPU baseline running on 32 threads.

	+Sampling		−Sampling	
	X	X+U	X	X+U
GPU one-by-one	7.23	4.7	2.39	0.71
CPU single-thread	12.7	4.8	0.32	0.05

Table 4: Sentence-by-sentence GPU grammar extraction throughput (words/second) vs. a single thread on the CPU (X: Xinhua, X+U: Xinhua + UN).

Queries	2k	4k	6k	8k
GPU PBMT	15s	25s	29s	33s
GPU Hiero	43s	73s	90s	107s
Slowdown	2.8×	2.9×	3.1×	3.2×

Table 5: Grammar extraction time comparing this work (GPU Hiero) and the work of He et al. (2013) (GPU PBMT).

to sampling (Callison-Burch et al., 2005; Lopez, 2008b), these results illustrate the raw computational potential of GPUs, showing that we can eliminate heuristics that make CPU processing tractable. We believe future work can exploit these untapped processing cycles to improve translation quality.

How does the GPU fare for translation tasks that demand low latency, such as sentence-by-sentence translation on the web? To find out, we conducted experiments where the sentences are fed, one by one, to the GPU grammar extraction algorithm. Results are shown in Table 4, with a comparison to a single-threaded CPU baseline. To be consistent with the other results, we also measure speed in terms of throughput here. Note that we are not aware of any freely available multi-threaded CPU algorithm to process an *individual* sentence in parallel, so the single-thread CPU comparison is reasonable.[3] We observe that the GPU is slower only with sampling on the smaller Xinhua data. In all other cases, sentence-by-sentence processing on the GPU achieves a similar level of performance or is faster.

[3]Parallel sub-sentential parsing has been known for many years (Chandwani et al., 1992) although we don't know of an implementation in any major open-source MT systems.

Next, we compare the performance of hierarchical grammar extraction to phrase-based extraction on the GPU with sampling. We replicated the test data condition of He et al. (2013) so that our first 8k query sentences are the same as those used in their experiments. The results are shown in Table 5, where we report grammar extraction time for batches of different sizes; the bottom row shows the slowdown of the hierarchical vs. non-hierarchical grammar conditions. This quantifies the performance penalty to achieve the translation quality gains reported in Table 1. Hierarchical grammar extraction is about three times slower, primarily due to the computational costs of the new passes presented in §2.

Another aspect of performance is memory footprint. We report the memory use (CPU RAM) of all four conditions in Table 6. The values reported for the CPU implementation use a single thread only. At runtime, our hierarchical GPU system exhibits peak CPU memory use of 8 GB on the host machine. Most of this memory is consumed by batching ex-

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

	CPU	GPU
PBMT	1.6 GB	2.3 GB
Hiero	1.9 GB	8.0 GB

Table 6: Memory consumption (CPU RAM) for different experimental conditions.

tracted phrases before scoring in passes 10 through 14. Since the phrase-based GPU implementation processes far fewer phrases, the memory footprint is much smaller. The CPU implementations process extracted phrases in small batches grouped by source phrase, and thus exhibit less memory usage. However, these levels of memory consumption are modest considering modern hardware. In all other respects, memory usage is similar for all systems, since the suffix array and associated data structures are all linear in the size of the indexed parallel text.

4.3 Per-Pass Speed

To obtain a detailed picture of where the GPU speedups and bottlenecks are, we collected per-pass timing statistics. Table 7 shows results for grammar extraction on 6k queries using the Xinhua data with no sampling and default length constraints (passes in gray occur on the GPU; all others on the CPU). These numbers explain the decreased speed of hierarchical extraction compared to He et al. (2013), with the new passes (shown in *italics*) accounting for more than 75% of the total computation time. However, even passes that are nominally the same actually require more time in the hierarchical case: in extracting and scoring phrases associated with a contiguous pattern u, we must now also extract and score patterns $\star\ u$, $u\ \star$, and $\star\ u\ \star$.

Interestingly, the CPU portions of our algorithm account for around half of the total grammar extraction time. One way to interpret this observation is that the massive parallelization provided by the GPU is so effective that we are bottlenecked by the CPU. In our current design, the CPU portions are those that cannot be easily parallelized on the GPU or those that require too much memory to fit on the GPU. The former is a possible target for optimization in future work, though the latter will likely be solved by hardware advances alone: for example, the Tesla K80 has 24 GB of memory.

Pass	Time	%
Contig. pattern pass 1	0.03	0.02%
Contig. pattern pass 2	0.02	0.02%
One-gap pattern generation	1.14	0.86%
One-gap pattern matching	19.24	14.54%
Two-gap pattern generation	0.37	0.28%
Two-gap pattern matching	15.91	12.02%
Gappy pattern processing	1.21	0.91%
Contig. pattern extraction	13.94	10.53%
Two-gap pattern extraction	0.52	0.39%
One-gap pattern extraction	11.07	8.36%
One gap translation features	23.72	17.91%
Two gap translation features	30.90	23.34%
Contig. translation features	5.00	3.77%
Lexical weight feature	3.09	2.33%
Data transfer and control	6.23	4.71%
Total	132.39	100.0%

Table 7: Detailed timings (in seconds) for 6k queries. Passes in gray occur on the GPU; all others on the CPU. Passes needed for hierarchical grammars are in *italics*, which are not present in He et al. (2013).

4.4 End-to-end Speed

What is the benefit of using GPUs in an end-to-end translation task? Since we have shown that both the CPU and GPU implementations achieve near-identical translation quality, the difference lies in speed. But speed is difficult to measure fairly: translation involves not only grammar extraction but also decoding and associated I/O. We have focused on grammar extraction on the GPU, but our research vision involves eventually moving all components of the machine translation pipeline onto the GPU. The experiments we describe below capture the performance advantages of our implementation that are achievable today, using cdec for decoding (on the CPU, using 32 threads).

To measure end-to-end translation speed using pycdec, per-sentence grammars are first extracted for a batch of sentences and written to disk, then read from disk during decoding. Therefore, we report times separately for grammar extraction, disk I/O, and decoding (which includes time for reading the grammar files from disk back into memory). Grammar extraction is either performed on the GPU

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

Grammar Extraction	Disk I/O	Decoding
GPU: 30.8s CPU: 101.1s	13.4s	CPU: 59s

Table 8: Running times for an end-to-end translation pipeline over NIST03 test data. Grammar extraction is either performed on the GPU or the CPU (32 threads); other stages are the same for both conditions (decoding uses 32 threads).

or on the CPU (using 32 threads), same as the experiments described in the previous section.

Results are shown in Table 8 using the Xinhua training data and NIST03 test data (919 sentences, 27,045 words). All experiment settings are exactly the same as in the previous section. We observe an end-to-end translation throughput of 262 words/second with GPU grammar extraction and 156 words/second on the CPU (32 threads), for a speedup of 1.68×.

Despite the speedup, we note that this experiment favors the CPU for several reasons. First, the GPU is idle during decoding, but it could be used to process grammars for a subsequent batch of sentences in a pipelined fashion. Second, NIST03 is a small batch that doesn't fully saturate the GPU—throughput keeps increasing by a large margin with larger batch sizes (see results in Table 3). Third, in comparison to the 32-thread CPU baseline, our GPU extraction only uses a single thread on the CPU throughout its execution, thus the CPU portion of the performance can be further improved, especially in the feature generation passes (see §4.3 for details).

Of course, the GPU/CPU combination requires a server equipped with a GPU, incurring additional hardware costs. We estimate that in Q4 2014 dollars, our base system would cost roughly $7500 USD, and the GPU would cost another $2600 USD. However, the server-grade GPU used in this work is not the only choice: a typical high-end consumer GPU, such as the NVIDIA GTX Titan Black (around $1100), costs considerably less but has even higher memory bandwidth and with similarly impressive floating point performance. This price difference is due to extra functionalities (e.g., error-correcting code memory) for specific applications (e.g., scientific computing), and is not directly related to differences in raw computational power. This means that we could speed up overall translation by 68% if we spend an additional 35% (server-grade GPU) or 15% (consumer-grade GPU) on hardware. From an economic perspective, this is an attractive proposition. Of course, the advantages of using GPUs for high-throughput translation go up further with larger batch sizes.

4.5 One-time Construction Costs

Construction of static data structures for on-demand grammar extraction is a one-time cost given a corpus T. However, under a streaming scenario where we might receive incremental changes to T as new training data become available, we need to update the data structures appropriately.

Updating static data structures involves two costs: the suffix array with its LCP array and the precomputation indexes. We do not consider the alignment construction cost as it is external to cdec (and is a necessary step for all implementations). For the Xinhua data, building the suffix array using a single CPU thread takes 29.2 seconds and building the precomputation indexes on the GPU takes 5.7 seconds. Compared to Table 7, these one-time costs represent approximately 27% of the GPU grammar extraction time.

It is possible to lower the construction costs of these data structures given recent advances. Levenberg et al. (2010) describe novel algorithms that allow efficient in-place updates of the suffix array when new training data arrive. That work directly tackles on-demand SMT architectures in the streaming data scenario. Alternatively, the speed of suffix array construction can be improved significantly by the CUDA Data Parallel Primitives Library,[4] which provides fast sorting algorithms to efficiently construct suffix arrays on the GPU. Minimizing data preparation costs has not been a focus of this work, but we believe that the massive parallelism provided by the GPU represents promising future work.

5 Conclusions and Future Work

The increasing demands for translation services because of globalization (Pangeanic, 2013; Sykes, 2009) make high-throughput translation a realistic

[4]http://cudpp.github.io/cudpp/2.2/

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

scenario, and one that our GPU implementation is highly suited to serve. High-throughput translation also enables downstream applications such as document translation in cross-language information retrieval (Oard and Hackett, 1997), where we translate the entire source document collection into the target language prior to indexing.

The number of transistors on a chip continues to increase exponentially, a trend that even pessimists concede should continue at least until the end of the decade (Vardi, 2014). Computer architects widely agree that instruction-level hardware parallelism is long past the point of diminishing returns (Olukotun and Hammond, 2005). This has led to a trend of placing greater numbers of cores on the same die. The question is how to best utilize the transistor budget: a small number of complex cores, a large number of simple cores, or a mixture of both? For our problem, it appears that we can take advantage of brute force scans and fine-grained parallelism inherent in the problem of on-demand extraction, which makes investments in large numbers of simple cores (as on the GPU) a win.

This observation is in line with trends in other areas of computing. Many problems in computational biology, like computational linguistics, boil down to efficient search on discrete sequences of symbols. DNA sequence alignment systems MummerGPU (Schatz et al., 2007) and MummerGPU 2 (Trapnell and Schatz, 2009) use suffix trees to perform DNA sequence matching on the GPU, while the state-of-the-art system MummurGPU++ (Gharaibeh and Ripeanu, 2010) uses suffix arrays, as we do here. Our algorithms for matching gappy patterns in passes 3 and 4 are closely related to seed-and-extend algorithms for approximate matching in DNA sequences, which have recently been implemented on GPUs (Wilton et al., 2014).

It is unlikely that CPU processing will become obsolete, since not all problems can be cast in a data-parallel framework. Ultimately, we need a hybrid architecture where parallelizable tasks are offloaded to the GPU, which works in conjunction with the CPU to handle irregular computations in a pipelined fashion. In a well-balanced system, both the GPU and the CPU would be fully utilized, performing the types of computation they excel at, unlike in our current design, where the GPU sits idle while the CPU finishes decoding. A part of our broad research agenda is exploring which aspects of the machine translation pipeline are amenable to GPU algorithms. The performance analysis in §4.3 shows that even in grammar extraction there are CPU bottlenecks we need to address and opportunities for further optimization.

Beyond grammar extraction, there is a question about whether decoding can be moved to the GPU. Memory is a big hurdle: since accessing data structures off-GPU is costly, it would be preferable to hold all models in GPU memory. We've addressed the problem for translation models, but the language models used in machine translation are also large. It might be possible to use lossy compression (Talbot and Osborne, 2007) or batch request strategies (Brants et al., 2007) to solve this problem. If we do, we believe that translation models could be decoded using variants of GPU algorithms for speech (Chong et al., 2009) or parsing (Yi et al., 2011; Canny et al., 2013; Hall et al., 2014), though the latter algorithms exploit properties of latent-variable grammars that may not extend to translation. Thinking beyond decoding, we believe that other problems in computational linguistics might benefit from the massive parallelism offered by GPUs.

Acknowledgments

This research was supported in part by the BOLT program of the Defense Advanced Research Projects Agency, Contract No. HR0011-12-C-0015; NSF under award IIS-1218043; and the Human Language Technology Center of Excellence at Johns Hopkins University. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect views of the sponsors. The second author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob. We thank Nikolay Bogoychev and the anonymous TACL reviewers for helpful comments on previous drafts, Rich Cox for support and advice, and CLIP labmates (particularly Junhui Li and Wu Ke) for helpful discussions. We also thank UMIACS for providing hardware resources via the NVIDIA CUDA Center of Excellence, and the UMIACS IT staff, especially Joe Webster, for excellent support.

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

References

Paul Baltescu and Phil Blunsom. 2014. A fast and simple online synchronous context free grammar extractor. *The Prague Bulletin of Mathematical Linguistics*, 102(1):17–26.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, pages 858–867.

Chris Callison-Burch, Colin Bannard, and Josh Schroeder. 2005. Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics (ACL 2005)*, pages 255–262.

John Canny, David Hall, and Dan Klein. 2013. A multi-teraflop constituency parser using GPUs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processin (EMNLP 2013)*, pages 1898–1907.

Victor Chahuneau, Noah A. Smith, and Chris Dyer. 2012. pycdec: A Python interface to cdec. In *Proceedings of the 7th Machine Translation Marathon (MTM 2012)*.

M. Chandwani, M. Puranik, and N. S. Chaudhari. 1992. On CKY-parsing of context-free grammars in parallel. In *Proceedings of Technology Enabling Tomorrow: Computers, Communications and Automation towards the 21st Century*, pages 141–145.

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

David Chiang. 2012. Hope and fear for discriminative training of statistical translation models. *Journal of Machine Learning Research*, 13:1159–1187.

Jike Chong, Ekaterina Gonina, Youngmin Yi, and Kurt Keutzer. 2009. A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit. In *Proceedings of the 10th Annual Conference of the International Speech Communication Association (INTERSPEECH 2009)*, pages 1183–1186.

Jonathan H. Clark, Chris Dyer, Alon Lavie, and Noah A. Smith. 2011. Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)*, pages 176–181.

Chris Dyer, Adam Lopez, Juri Ganitkevitch, Johnathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. 2010. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12.

Michel Galley and Christopher D. Manning. 2010. Accurate non-hierarchical phrase-based translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL 2010)*, pages 966–974.

Abdullah Gharaibeh and Matei Ripeanu. 2010. Size matters: Space/time tradeoffs to improve GPGPU applications performance. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1–12.

Jesús González-Rubio, Daniel Ortiz-Martínez, and Francisco Casacuberta. 2012. Active learning for interactive machine translation. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2012)*, pages 245–254.

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

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.

Jared Hoberock and Nathan Bell. 2010. Thrust: A parallel template library. Version 1.8.0.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 177–180.

Abby Levenberg, Chris Callison-Burch, and Miles Osborne. 2010. Stream-based translation models for statistical machine translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL 2010)*, pages 394–402.

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024

Adam Lopez. 2007. Hierarchical phrase-based translation with suffix arrays. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, pages 976–985.

Adam Lopez. 2008a. *Machine translation by pattern matching*. Ph.D. dissertation, University of Maryland, College Park, Maryland, USA.

Adam Lopez. 2008b. Tera-scale translation models via pattern matching. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING 2008)*, pages 505–512.

Udi Manber and Gene Myers. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 319–327.

Douglas W. Oard and Paul Hackett. 1997. Document translation for cross-language text retrieval at the University of Maryland. In *Proceedings of the 7th Text REtrieval Conference (TREC-7)*.

Franz Josef Och, Christoph Tillmann, and Hermann Ney. 1999. Improved alignment models for statistical machine translation. In *Proceedings of the 1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP 1999)*, pages 20–28.

Kunle Olukotun and Lance Hammond. 2005. The future of microprocessors. *ACM Queue*, 3(7):27–34.

Pangeanic, 2013. *What is The Size of the Translation Industry?* http://www.pangeanic.com/knowledge_center/size-translation-industry/.

Aaron B. Philips. 2012. *Modeling Relevance in Statistical Machine Translation: Scoring Alignment, Context, and Annotations of Translation Instances*. Ph.D. thesis, Carnegie Mellon University.

Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. 2007. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474.

Michel Simard, Nicola Cancedda, Bruno Cavestro, Marc Dymetman, Eric Gaussier, Cyril Goutte, and Kenji Yamada. 2005. Translating with non-contiguous phrases. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (EMNLP 2005)*, pages 755–762.

Patrick Simianer, Stefan Riezler, and Chris Dyer. 2012. Joint feature selection in distributed stochastic learning for large-scale discriminative training in SMT. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2012)*, pages 11–21.

Tanisha Sykes, 2009. *Growth in Translation*. http://www.inc.com/articles/2009/08/translation.html.

David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL 2007)*, pages 512–519.

Cole Trapnell and Michael C. Schatz. 2009. Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing*, 35(8-9):429–440.

Moshe Y. Vardi. 2014. Moore's Law and the sand-heap paradox. *Communications of the ACM*, 57(5):5.

Richard Wilton, Tamas Budavari, Ben Langmead, Sarah Wheelan, Steven L. Salzberg, and Alex Szalay. 2014. Faster sequence alignment through GPU-accelerated restriction of the seed-and-extend search space. http://dx.doi.org/10.1101/007641.

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.

Ying Zhang and Stephan Vogel. 2005. An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *Proceedings of the Tenth Conference of the European Association for Machine Translation (EAMT-05)*.

Downloaded from http://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00124/1566756/tacl_a_00124.pdf by guest on 13 October 2024