Massively Parallel Suffix Array Queries and On-Demand Phrase Extraction for Statistical Machine Translation Using GPUs

Massively Parallel Suffix Array Queries and On-Demand Phrase Extraction for Statistical Machine Translation Using GPUs

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

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

Adam Lopez
HLTCOE
Johns Hopkins University
Baltimore, Maryland
alopez@cs.jhu.edu

Abstract

Translation models in statistical machine translation can be scaled to large corpora and arbitrarily-long phrases by looking up translations of source phrases "on the fly" in an indexed parallel corpus using suffix arrays. However, this can be slow because on-demand extraction of phrase tables is computationally expensive. We address this problem by developing novel algorithms for general purpose graphics processing units (GPUs), which enable suffix array queries for phrase lookup and phrase extraction to be massively parallelized. Compared to a highly-optimized, state-of-the-art serial CPU-based implementation, our techniques achieve at least an order of magnitude improvement in terms of throughput. This work demonstrates the promise of massively parallel architectures and the potential of GPUs for tackling computationally-demanding problems in statistical machine translation and language processing.

1 Introduction

Efficiently handling large translation models is a perennial problem in statistical machine translation. One particularly promising solution (§2) is to use the parallel text itself as an *implicit* representation of the translation model and extract translation units "on the fly" when they are needed to decode new input (Brown, 2004). This idea has been applied to phrase-based (Callison-Burch et al., 2005; Zhang and Vogel, 2005), hierarchical (Lopez, 2007; Lopez, 2008b; Lopez, 2008a), and syntax-based (Cromieres and Kurohashi, 2011) models. A benefit of this technique is that it scales to arbitrarily large models with very little pre-processing. For instance, Lopez (2008b) showed that a translation model trained on a large corpus with sparse word alignments and loose extraction heuristics substantially improved Chinese-English translation. An explicit representation of the model would have required nearly a terabyte of memory, but its implicit representation using the parallel text required only a few gigabytes.

Unfortunately, there is substantial computational cost in searching a parallel corpus for source phrases, extracting their translations, and scoring them on the fly. Since the number of possible translation units may be quite large (for example, all substrings of a source sentence) and their translations are numerous, both phrase lookup and extraction are performance bottlenecks. Despite considerable research and the use of efficient indexes like suffix arrays (Manber and Myers, 1990), this problem remains not fully solved.

We show how to exploit the massive parallelism offered by modern general purpose graphics processing units (GPUs) to eliminate the computational bottlenecks associated with "on the fly" phrase extraction. GPUs have previously been applied to DNA sequence matching using suffix trees (Schatz et al., 2007) and suffix arrays (Gharaibeh and Ripeanu, 2010). Building on this work, we present two novel contributions: First, we describe improved GPU algorithms for suffix array queries that achieve greater parallelism (§3). Second, we propose novel data structures and algorithms for phrase extraction (§4) and scoring (§5) that are amenable to GPU par-

Proceedings of NAACL-HLT 2013, pages 325–334,
Atlanta, Georgia, 9–14 June 2013.

allelization. The resulting implementation achieves at least an order of magnitude higher throughput than a state-of-the-art single-threaded CPU implementation (§6). Since our experiments verify that the GPU implementation produces *exactly* the same results as a CPU reference implementation on a full extraction, we can simply replace that component and reap significant performance advantages with no impact on translation quality. To the best of our knowledge, this is the first reported application of GPU acceleration techniques for statistical machine translation. We believe these results reveal a promising yet unexplored future direction in exploiting parallelism to tackle perennial performance bottlenecks in state-of-the-art translation models.

2 Phrase Extraction On Demand

Lopez (2008b) provides the following recipe for "translation by pattern matching", which we use as a guide for the remainder of this paper:

Algorithm 1 Translation by pattern matching

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

The computational bottleneck occurs in lines 2–7: there are vast numbers of query phrases, matching occurrences, and extracted phrase pairs to process in the loops. In the next three sections, we attack each problem in turn.

3 Finding Every Phrase

First, we must find all occurrences of each source phrase in the input (line 3, Algorithm 1). This is a classic application of string pattern matching: given a short query pattern, the task is to find all occurrences in a much larger text. Solving the problem efficiently is crucial: for an input sentence F of length $|F|$, each of its $O(|F|^2)$ substrings is a potential query pattern.

3.1 Pattern Matching with Suffix Arrays

Although there are many algorithms for pattern matching, all of the examples that we are aware of for machine translation rely on *suffix arrays*. We briefly review the classic algorithms of Manber and Myers (1990) here since they form the basis of our techniques and analysis, but readers who are familiar with them can safely skip ahead to additional optimizations (§3.2).

A suffix array represents all suffixes of a corpus in lexicographical order. Formally, for a text T, the ith suffix of T is the substring of the text beginning at position i and continuing to the end of T. Each suffix can therefore be uniquely identified by the index i of its first word. A suffix array $S(T)$ of T is a permutation of these suffix identifiers $[1, |T|]$ arranged by the lexicographical order of the corresponding suffixes—in other words, the suffix array represents a sorted list of all suffixes in T. With both T and $S(T)$ in memory, we can find any query pattern Q in $O(|Q| \log |T|)$ time by comparing pattern Q against the first $|Q|$ characters of up to $\log |T|$ different suffixes using binary search.

An inefficiency in this solution is that each comparison in the binary search algorithm requires comparing all $|Q|$ characters of the query pattern against some suffix of text T. We can improve on this using an observation about the *longest common prefix* (LCP) of the query pattern and the suffix against which it is compared. Suppose we search for a query pattern Q in the span of the suffix array beginning at suffix L and ending at suffix R. For any suffix M which falls lexicographically between those at L and R, the LCP of Q and M will be at least as long as the LCP of Q and L or Q and R. Hence if we know the quantity $h = \textsc{Min}(\textsc{Lcp}(Q, L), \textsc{Lcp}(Q, R))$ we can skip comparisons of the first h symbols between Q and the suffix M, since they must be the same.

The solution of Manber and Myers (1990) exploits this fact along with the observation that each comparison in binary search is carried out according to a fixed recursion scheme: a query is only ever compared against a specific suffix M for a single range of suffixes bounded by some fixed L and R. Hence if we know the longest common prefix between M and each of its corresponding L and R according to the fixed recursions in the

algorithm, we can maintain a bound on h and reduce the aggregate number of symbol comparisons to $O(|Q| + \log |T|)$. To accomplish this, in addition to the suffix array, we pre-compute two other arrays of size $|T|$ for both left and right recursions (called the LCP arrays).

Memory use is an important consideration, since GPUs have less memory than CPUs. For the algorithms described here, we require four arrays: the original text T, the suffix array $S(T)$, and the two LCP arrays. We use a representation of T in which each word has been converted to a unique integer identifier; with 32-bit integers the total number of bytes is $16|T|$. As we will show, this turns out to be quite modest, even for large parallel corpora (§6).

3.2 Suffix Array Efficiency Tricks

Previous work on translation by pattern matching using suffix arrays on serial architectures has produced a number of efficiency optimizations:

1. Binary search bounds for longer substrings are initialized to the bounds of their longest prefix. Substrings are queried only if their longest prefix string was matched in the text.
2. In addition to conditioning on the longest prefix, Zhang and Vogel (2005) and Lopez (2007) condition on a successful query for the longest proper suffix.
3. Lopez (2007) queries each unique substring of a sentence exactly once, regardless of how many times it appears in an input sentence.
4. Lopez (2007) directly indexes one-word substrings with a small auxiliary array, so that their positions in the suffix array can be found in constant time. For longer substrings, this optimization reduces the $\log |T|$ term of query complexity to $\log(\text{count}(a))$, where a is the first word of the query string.

Although these efficiency tricks are important in the serial algorithms that serve as our baseline, not all of them are applicable to parallel architectures. In particular, optimizations (1), (2), and (3) introduce order dependencies between queries; they are disregarded in our GPU implementation so that we can fully exploit parallelization opportunities. We have not yet fully implemented (4), which is orthogonal to parallelization: this is left for future work.

3.3 Finding Every Phrase on a GPU

Recent work in computational biology has shown that suffix arrays are particularly amenable to GPU acceleration: the suffix-array-based DNA sequence matching system MummurGPU++ (Gharaibeh and Ripeanu, 2010) has been reported to outperform the already fast MummurGPU 2 (Trapnell and Schatz, 2009), based on suffix trees (an alternative indexing structure). Here, we apply the same ideas to machine translation, introducing some novel improvements to their algorithms in the process.

A natural approach to parallelism is to perform all substring queries in parallel (Gharaibeh and Ripeanu, 2010). There are no dependencies between iterations of the loop beginning on line 2 of Algorithm 1, so for input sentence F, we can parallelize by searching for all $O(|F|^2)$ substrings concurrently. We adopt this approach here.

However, naïve application of query-level parallelism leads to a large number of wasted threads, since most long substrings of an input sentence will not be found in the text. Therefore, we employ a novel two-pass strategy: in the first pass, we simply compute, for each position i in the input sentence, the length j of the longest substring in F that appears in T. These computations are carried out concurrently for every position i. During this pass, we also compute the suffix array bounds of the one-word substring $F[i]$, to be used as input to the second pass—a variant of optimizations (1) and (4) discussed in §3.2. On the second pass, we search for all substrings $F[i, k]$ for all $k \in [i + 1, i + j]$. These computations are carried out concurrently for all substrings longer than one word.

Even more parallelization is possible. As we saw in §3.1, each query in a suffix array actually requires two binary searches: one each for the first and last match in $S(T)$. The abundance of inexpensive threads on a GPU permits us to perform both queries concurrently on separate threads. By doing this in both passes we utilize more of the GPU's processing power and obtain further speedups.

As a simple example, consider an input sentence "The government puts more tax on its citizens", and suppose that substrings "The government", "government puts", and "puts more tax" are found in the training text, while none of the words in "on

Initial Word	Longest Match	Substrings	Threads	
			1st pass	2nd pass
The	2	*The, The government*	2	2
government	2	*government, government puts*	2	2
puts	3	*puts, puts more, puts more tax*	2	4
more	2	*more, more tax*	2	2
tax	1	*tax*	2	0
on	0	–	2	0
its	0	–	2	0
citizens	0	–	2	0
Total Threads:			16	10

Table 1: Example of how large numbers of suffix array queries can be factored across two highly parallel passes on a GPU with a total of 26 threads to perform all queries for this sample input sentence.

its citizens" are found. The number of threads spawned is shown in Table 1: all threads during a pass execute in parallel, and each thread performs a binary search which takes no more than $O(|Q| + \log |T|)$ time. While spawning so many threads may seem wasteful, this degree of parallelization still under-utilizes the GPU; the hardware we use (§6) can manage up to 21,504 concurrent threads in its resident occupancy. To fully take advantage of the processing power, we process multiple input sentences in parallel. Compared with previous algorithms, our two-pass approach and our strategy of thread assignment to increase the amount of parallelism represent novel contributions.

4 Extracting Aligned Target Phrases

The problem at line 5 of Algorithm 1 is to extract the target phrase aligned to each matching source phrase instance. Efficiency is crucial since some source phrases occur hundreds of thousands of times.

Phrase extraction from word alignments typically uses the consistency check of Och et al. (1999). A *consistent* phrase is one for which no words inside the phrase pair are aligned to words outside the phrase pair. Usually, consistent pairs are computed offline via dynamic programming over the alignment grid, from which we extract all consistent phrase pairs up to a heuristic bound on phrase length.

The online extraction algorithm of Lopez (2008a) checks for consistent phrases in a different manner. Rather than finding all consistent phrase pairs in a sentence, the algorithm asks: given a specific source phrase, is there a consistent phrase pair

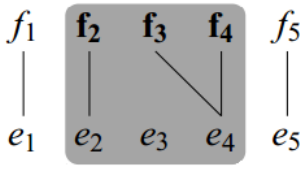

Figure 1: Source phrase $f_2 f_3 f_4$ and target phrase $e_2 e_3 e_4$ are extracted as a consistent pair, since the back-projection is contained within the original source span.

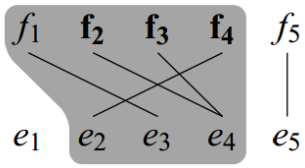

Figure 2: Source phrase $f_2 f_3 f_4$ and target phrase $e_2 e_3 e_4$ should not be extracted, since the back-projection is *not* contained within the original source span.

of which it is one side? To answer this, it first computes the projection of the source phrase in the target sentence: the minimum span containing all words that are aligned to any word of the source span. It then computes the projection of the target span back into the source; if this back-projection is contained within the original source span, the phrase pair is consistent, and the target span is extracted as the translation of the source. Figure 1 shows a "good" pair for source phrase $f_2 f_3 f_4$, since the back-projection is contained within the original source span, whereas Figure 2 shows a "bad" pair for source phrase $f_2 f_3 f_4$ since the back-projection is *not* contained within the original source span.

4.1 Sampling Consistent Phrases

Regardless of how efficient the extraction of a single target phrase is made, the fact remains that there are many phrases to extract. For example, in our Chinese Xinhua dataset (see §6), from 8,000 input query sentences, about 20 million source substrings can be extracted. The standard solution to this problem is to sample a set of occurrences of each source phrase, and only extract translations for those occurrences (Callison-Burch et al., 2005; Zhang and Vogel, 2005). As a practical matter, this can be done by sampling at uniform intervals from the matching span of a suffix array. Lopez (2008a) reports a sample size of 300; for phrases occurring fewer than 300 times, all translations are extracted.

4.2 GPU Implementation

We present novel data structures and an algorithm for efficient phrase extraction, which together are amenable to massive parallelization on GPUs. The basic insight is to pre-compute data structures for the source-to-target alignment projection and back-projection procedure described by Lopez (2008a) for checking consistent alignments.

Let us consider a single matching substring (from the output of the suffix array queries), span $[i, j]$ in the source text T. For each k, we need to know the leftmost and rightmost positions that it aligns to in the target T'. For this purpose we can define the target span $[i', j']$, along with leftmost and rightmost arrays L and R as follows:

$$i' := \min_{k \in [i,j]} L(k)$$
$$j' := \max_{k \in [i,j]} R(k)$$

The arrays L and R are each of length $|T|$, indexed by absolute corpus position. Each array element contains the leftmost and rightmost extents of the source-to-target alignments (in the target), respectively. Note that in order to save space, the values stored in the arrays are sentence-relative positions (e.g., token count from the beginning of each sentence), so that we only need one byte per array entry. Thus, i' and j' are sentence-relative positions (in the target).

Similarly, for the back-projection, we use two arrays L' and R' on the target side (length $|T'|$) to keep track of the leftmost and rightmost positions that k' in the target training text align to, as below:

$$i'' := \min_{k' \in [s'+i', s'+j']} L'(k')$$
$$j'' := \max_{k' \in [s'+i', s'+j']} R'(k')$$

The arrays L' and R' are indexed by absolute corpus positions, but their contents are sentence relative positions (on the source side). To index the arrays L' and R', we also need to obtain the corresponding target sentence start position s'. Note that the back-projected span $[i'', j'']$ may or may not be the same as the original span $[i, j]$. In fact, this is exactly what we must check for to ensure a consistent alignment.

The suffix array gives us i, which is an absolute corpus position, but we need to know the sentence-relative position, since the spans computed by R, L, R', L' are all sentence relative. To solve this, we introduce an array P (length $|T|$) that gives the relative sentence position of each source word.

We then pack the three source side arrays (R, L, and P) into a single RLP array of 32-bit integers (note that we are actually wasting one byte per array element). Finally, since the end-of-sentence special token is not used in any of R, L, or P, its position in RLP can be used to store an index to the start of the corresponding target sentence in the target array T'. Now, given a source phrase spanning $[i, j]$ (recall, these are absolute corpus positions), our phrase extraction algorithm is as follows:

Algorithm 2 Efficient Phrase Extraction Algorithm

1: **for each** source span $[i, j]$ **do**
2: Compute $[i', j']$
3: $s := i - P[i] - 1$
4: $s' := RLP[s]$
5: $i'' := \min_{k' \in [s'+i', s'+j']} L'(k')$
6: $j'' := \max_{k' \in [s'+i', s'+j']} R'(k')$
7: If $i - s = i''$ and $j - s = j''$ then
8: Extract $T[i, j]$ with $T'[s' + i', s' + j']$

where s is the source sentence start position of a given source phrase and s' is the target sentence start position. If the back-projected spans match the original spans, the phrase pair $T[i, j]$ and $T'[s' + i', s' + j']$ is extracted.

In total, the data structures RLP, R', and L' require $4|T| + 2|T'|$ bytes. Not only is this phrase

extraction algorithm fast—requiring only a few indirect array references—the space requirements for the auxiliary data structures are quite modest.

Given sufficient resources, we would ideally parallelize the phrase table creation process for each occurrence of the matched source substring. However, the typical number of source substring matches for an input sentence is even larger than the number of threads available on GPUs, so this strategy does not make sense due to context switching overhead. Instead, GPU thread blocks (groups of 512 threads) are used to process each source substring. This means that for substrings with large numbers of matches, one thread in the GPU block would process multiple occurrences. This strategy is widely used, and according to GPU programming best practices from NVIDIA, allocating more work to a single thread maintains high GPU utilization and reduces the cost of context switches.

5 Computing Every Feature

Finally, we arrive at line 7 in Algorithm 3, where we must compute feature values for each extracted phrase pair. Following the implementation of grammar extraction used in cdec (Lopez, 2008a), we compute several widely-used features:

1. Pair count feature, $c(e, f)$.
2. The joint probability of all target-to-source phrase translation probabilities, $p(e|f) = c(e, f)/c(f)$, where e is target phrase, f is the source phrase.
3. The logarithm of the target-to-source lexical weighting feature.
4. The logarithm of the source-to-target lexical weighting feature.
5. The coherence probability, defined as the ratio between the number of successful extractions of a source phrase to the total count of the source phrase in the suffix array.

The output of our phrase extraction is a large collection of phrase pairs. To extract the above features, aggregate statistics need to be computed over phrase pairs. To make the solution both compact and efficient, we first sort the unordered collection of phrases from the GPU into an array, then the aggregate statistics can be obtained in a single pass over the array, since identical phrase pairs are now grouped together.

6 Experimental Setup

We tested our GPU-based grammar extraction implementation under the conditions in which it would be used for a Chinese-to-English machine translation task, in particular, replicating the data conditions of Lopez (2008b). Experiments were performed on two data sets. First, we used the source (Chinese) side of news articles collected from the Xinhua Agency, with around 27 million words of Chinese in around one million sentences (totaling 137 MB). Second, we added source-side parallel text from the United Nations, with around 81 million words of Chinese in around four million sentences (totaling 561 MB). In a pre-processing phase, we mapped every word to a unique integer, with two special integers representing end-of-sentence and end-of-corpus, respectively.

Input query data consisted of all sentences from the NIST 2002–2006 translation campaigns, tokenized and integerized identically to the training data. On average, sentences contained around 29 words. In order to fully stress our GPU algorithms, we ran tests on batches of 2,000, 4,000, 6,000, 8,000, and 16,000 sentences. Since there are only around 8,000 test sentences in the NIST data, we simply duplicated the test data as necessary.

Our experiments used NVIDIA's Tesla C2050 GPU (Fermi Generation), which has 448 CUDA cores with a peak memory bandwidth 144 GB/s. Note that the GPU was released in early 2010 and represents previous generation technology. NVIDIA's current GPUs (Kepler) boasts raw processing power in the 1.3 TFlops (double precision) range, which is approximately three times the GPU we used. Our CPU is a 3.33 GHz Intel Xeon X5260 processor, which has two cores.

As a baseline, we compared against the publicly available implementation of the CPU-based algorithms described by Lopez (2008a) found in the pycdec (Chahuneau et al., 2012) extension of the cdec machine translation system (Dyer et al., 2010). Note that we only tested grammar extraction for continuous pairs of phrases, and we did not test the slower and more complex queries for hierarchical

Input Sentences Number of Words		2,000 57,868	4,000 117,854	6,000 161,883	8,000 214,246	16,000 428,492
Xinhua						
With Sampling (s_{300})	GPU (words/second)	3811 (21.9)	4723 (20.4)	5496 (32.1)	6391 (29.7)	12405 (36.0)
	CPU (words/second)	200 (1.5)				
	Speedup	**19**×	**24**×	**27**×	**32**×	**62**×
No Sampling (s_{∞})	GPU (words/second)	1917 (8.5)	2859 (11.1)	3496 (19.9)	4171 (23.2)	8186 (27.6)
	CPU (words/second)	1.13 (0.02)				
	Speedup	**1690**×	**2520**×	**3082**×	**3677**×	**7217**×
Xinhua + UN						
With Sampling (s_{300})	GPU (words/second)	2021 (5.3)	2558 (10.7)	2933 (13.9)	3439 (15.2)	6737 (29.0)
	CPU (words/second)	157 (1.8)				
	Speedup	**13**×	**16**×	**19**×	**22**×	**43**×
No Sampling (s_{∞})	GPU (words/second)	500.5 (2.5)	770.1 (3.9)	984.6 (5.8)	1243.8 (5.4)	2472.3 (12.0)
	CPU (words/second)	0.23 (0.002)				
	Speedup	**2194**×	**3375**×	**4315**×	**5451**×	**10836**×

Table 2: Comparing the GPU and CPU implementations for phrase extraction on two different corpora. Throughput is measured in words per second under different test set sizes; the 95% confidence intervals across five trials are given in parentheses, along with relative speedups comparing the two implementations.

(gappy) patterns described by Lopez (2007). Both our implementation and the baseline are written primarily in C/C++.[1]

Our source corpora and test data are the same as that presented in Lopez (2008b), and using the CPU implementation as a reference enabled us to confirm that our extracted grammars and features are identical (modulo sampling). We timed our GPU implementation as follows: from the loading of query sentences, extractions of substrings and grammar rules, until all grammars for all sentences are generated in memory. Timing does not include offline preparations such as the construction of the suffix array on source texts and the I/O costs for writing the per-sentence grammar files to disk. This timing procedure is exactly the same for the CPU baseline. We are confident that our results represent a fair comparison between the GPU and CPU, and are not attributed to misconfigurations or other flaws in experimental procedures. Note that the CPU implementation runs in a single thread, on the same machine that hosts the GPU (described above).

7 Results

Table 2 shows performance results comparing our GPU implementation against the reference CPU implementation for phrase extraction. In one experimental condition, the sampling parameter for frequently-matching phrases is set to 300, per Lopez (2008a), denoted s_{300}. The experimental condition without sampling is denoted s_{∞}. Following standard settings, the maximum length of the source phrase is set to 5 and the maximum length of the target phrase is set to 15 (same for both GPU and CPU implementations). The table is divided into two sections: the top shows results on the Xinhua data, and the bottom on Xinhua + UN data. Columns report results for different numbers

[1]The Chahuneau et al. (2012) implementation is in Cython, a language for building Python applications with performance-critical components in C. In particular, all of the suffix array code that we instrumented for these experiments are 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 from the original version.

# Sent.	2000	4000	6000	8000	16000
Speedup	9.6×	14.3×	17.5×	20.9×	40.9×
Phrases	2.1×	1.8×	1.7×	1.6×	1.6×

Table 3: Comparing no sampling on the GPU with sampling on the CPU in terms of performance improvements (GPU over CPU) and increases in the number of phrase pairs extracted (GPU over CPU).

of input sentences. Performance is reported in terms of throughput: the number of processed words per second on average (i.e., total time divided by the batch size in words). The results are averaged over five trials, with 95% confidence intervals shown in parentheses. Note that as the batch size increases, we achieve higher throughput on the GPU since we are better saturating its full processing power. In contrast, performance is constant on the CPU regardless of the number of sentences processed.

The CPU throughput on the Xinhua data is 1.13 words per second without sampling and 200 words per second with sampling. On 16,000 test sentences, we have mostly saturated the GPU's processing power, and observe a 7217× speedup over the CPU implementation without sampling and 62× speedup with sampling. On the larger (Xinhua + UN) corpus, we observe 43× and 10836× speedup with sampling and no sampling, respectively.

Interestingly, a run without sampling on the GPU is still substantially faster than a run with sampling on the CPU. On the Xinhua corpus, we observe speedups ranging from nine times to forty times, as shown in Table 3. Without sampling, we are able to extract up to twice as many phrases.

In previous CPU implementations of on-the-fly phrase extraction, restrictions were placed on the maximum length of the source and target phrases due to computational constraints (in addition to sampling). Given the massive parallelism afforded by the GPU, might we be able to lift these restrictions and construct the complete phrase table? To answer this question, we performed an experiment without sampling and without any restrictions on the length of the extracted phrases. The complete phrase table contained about 0.5% more distinct pairs, with negligible impact on performance.

When considering these results, an astute reader might note that we are comparing performance of a single-threaded implementation with a fully-saturated GPU. To address this concern, we conducted an experiment using a multi-threaded version of the CPU reference implementation to take full advantage of multiple cores on the CPU (by specifying the `-j` option in cdec); we experimented with up to four threads to fully saturate the dual-core CPU. In terms of throughput, the CPU implementation scales linearly, i.e., running on four threads achieves roughly 4× throughput. Note that the CPU and GPU implementations take advantage of parallelism in completely different ways: cdec can be characterized as embarrassingly parallel, with different threads processing each complete sentence in isolation, whereas our GPU implementation achieves intra-sentential parallelism by exploiting many threads to concurrently process each sentence. In terms of absolute performance figures, even with the 4× throughput improvement from fully saturating the CPU, our GPU implementation remains faster by a wide margin. Note that neither our GPU nor CPU represents state-of-the-art hardware, and we would expect the performance advantage of GPUs to be even greater with latest generation hardware, since the number of available threads on a GPU is increasing faster than the number of threads available on a CPU.

Since phrase extraction is only one part of an end-to-end machine translation system, it makes sense to examine the overall performance of the entire translation pipeline. For this experiment, we used our GPU implementation for phrase extraction, serialized the grammar files to disk, and used cdec for decoding (on the CPU). The comparison condition used cdec for all three stages. We used standard phrase length constraints (5 on source side, 15 on target side) with sampling of frequent phrases. Finally, we replicated the data conditions in Lopez (2008a), where our source corpora was the Xinhua data set and our development/test sets were the NIST03/NIST05 data; the NIST05 test set contains 1,082 sentences.

Performance results for end-to-end translation are shown in Table 4, broken down in terms of total amount of time for each of the processing stages for the entire test set under different conditions. In the decoding stage, we varied the number of CPU threads (note here we do not observe linear

Phrase Extraction	I/O	Decoding	
GPU: 11.0	3.7	1 thread	55.7
		2 threads	35.3
CPU: 166.5		3 threads	31.5
		4 threads	26.2

Table 4: End-to-end machine translation performance: time to process the NIST05 test set in seconds, broken down in terms of the three processing stages.

speedup). In terms of end-to-end results, complete translation of the test set takes 41 seconds with the GPU for phrase extraction and CPU for decoding, compared to 196 seconds using the CPU for both (with four decoding threads in both cases). This represents a speedup of 4.8×, which suggests that even selective optimizations of individual components in the MT pipeline using GPUs can make a substantial difference in overall performance.

8 Future Work

There are a number of directions that we have identified for future work. For computational efficiency reasons, previous implementations of the "translation by pattern matching" approach have had to introduce approximations, e.g., sampling and constraints on phrase lengths. Our results show that the massive amounts of parallelism available in the GPU make these approximations unnecessary, but it is unclear to what extent they impact translation quality. For example, Table 3 shows that we extract up to twice as many phrase pairs without sampling, but do these pairs actually matter? We have begun to examine the impact of various settings on translation quality and have observed small improvements in some cases (which, note, come for "free"), but so far the results have not been conclusive.

The experiments in this paper focus primarily on throughput, but for large classes of applications latency is also important. One current limitation of our work is that large batch sizes are necessary to fully utilize the available processing power of the GPU. This and other properties of the GPU, such as the high latency involved in transferring data from main memory to GPU memory, make low-latency processing a challenge, which we hope to address.

Another broad future direction is to "GPU-ify" other machine translation models and other components in the machine translation pipeline. An obvious next step is to extend our work to the hierarchical phrase-based translation model (Chiang, 2007), which would involve extracting "gappy" phrases. Lopez (2008a) has tackled this problem on the CPU, but it is unclear to what extent the same types of algorithms he proposed can execute efficiently in the GPU environment. Beyond phrase extraction, it might be possible to perform decoding itself in the GPU—not only will this exploit massive amounts of parallelism, but also reduce costs in moving data to and from the GPU memory.

9 Conclusion

GPU parallelism offers many promises for practical and efficient implementations of language processing systems. This promise has been demonstrated for speech recognition (Chong et al., 2008; Chong et al., 2009) and parsing (Yi et al., 2011), and we have demonstrated here that it extends to machine translation as well. We believe that explorations of modern parallel hardware architectures is a fertile area of research: the field has only begun to examine the possibilities and there remain many more interesting questions to tackle. Parallelism is critical not only from the perspective of building real-world applications, but for overcoming fundamental computational bottlenecks associated with models that researchers are developing today.

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-1144034. 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 would like to thank three anonymous reviewers for providing helpful suggestions and also acknowledge Benjamin Van Durme and CLIP labmates for useful discussions. We also thank UMIACS for providing hardware resources via the NVIDIA CUDA Center of Excellence, UMIACS IT staff, especially Joe Webster, for excellent support.

References

R. D. Brown. 2004. A modified Burrows-Wheeler Transform for highly-scalable example-based translation. In *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas (AMTA 2004)*, pages 27–36.

C. Callison-Burch, C. Bannard, and J. 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.

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

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

J. Chong, Y. Yi, A. Faria, N. R. Satish, and K. Keutzer. 2008. Data-parallel large vocabulary continuous speech recognition on graphics processors. In *Proceedings of the Workshop on Emerging Applications and Manycore Architectures*.

J. Chong, E. Gonina, Y. Yi, and K. 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.

F. Cromieres and S. Kurohashi. 2011. Efficient retrieval of tree translation examples for syntax-based machine translation. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP 2011*, pages 508–518.

C. Dyer, A. Lopez, J. Ganitkevitch, J. Weese, F. Ture, P. Blunsom, H. Setiawan, V. Eidelman, and P. 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.

A. Gharaibeh and M. 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.

A. 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*, pages 976–985.

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

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

U. Manber and G. 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.

F. J. Och, C. Tillmann, and H. 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*, pages 20–28.

M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. 2007. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474.

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

Y. Yi, C.-Y. Lai, S. Petrov, and K. Keutzer. 2011. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 175–185.

Y. Zhang and S. 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)*.