THE UNIVERSITY
of EDINBURGH

Fast machine translation on parallel and massively parallel hardware

Nikolay Bogoychev

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2019

Abstract

Parallel systems have been widely adopted in the field of machine translation, because the raw computational power they offer is well suited to this computationally intensive task. However programming for parallel hardware is not trivial as it requires redesign of the existing algorithms. In my thesis I design efficient algorithms for machine translation on parallel hardware. I identify memory accesses as the biggest bottleneck to processing speed and propose novel algorithms that minimize them. I present three distinct case studies in which minimizing memory access substantially improves speed: Starting with statistical machine translation, I design a phrase table that makes decoding ten times faster on a multi-threaded CPU. Next, I design a GPU-based n-gram language model that is twice as fast per £ as a highly optimized CPU implementation. Turning to neural machine translation, I design new stochastic gradient descent techniques that make end-to-end training twice as fast. The work in this thesis has been incorporated in two popular machine translation toolkits: Moses and Marian.

Lay Summary

Machine translation systems, such as Google translate and Microsoft translator, Baidu fanyi, etc, have become increasingly more popular in the recent years. Machines translate more than one billion words daily, however producing said translations is quite computationally expensive and requires hundreds of millions of computer chips working around the clock. In my thesis, I explain why machine translation systems require so much computational power. I also carefully examine the characteristics of modern computer chips and note that often the speed of memory is the limiting factor for achieving faster performance. I use that knowledge to design algorithms that can make machine translation much more efficient on modern hardware, increasing translation speed up to ten fold compared to previous state of the art work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

May 17, 2019 (*Nikolay Bogoychev*)

Acknowledgements

PhD has been long journey filled with unexpected twists. My field changed completely in my first year of study, I ended up living in a few different countries, experienced multiple highs and lows met a many wonderful people, learned a fair bit of new things and grew up into a better person... I struggled for many sleepless nights, wondering whether I would finish this at all. Fortunately I got plenty of emotional support from my supervisors and my peers and above all I understood that a PhD is about learning, perseverance and continuous self improvement. Every rejection, however painful, should also a motivation to do better in the future. I did not learn that by myself, but with ample help from supervisors, coworkers and friends I accepted it. I want to thank my precious office mates, past and present: Joana, Federico, Maria, Mihaela, Vanya, Jason (whose name I kept forgetting), Sefa, Esma, Chao, Aurora, Andrew, Ratish for all the laughter, support and motivation they brought me. I want to thank coworkers around the forum: Carol, Craig, Clara, Ida, John, Matt, Naomi, Pippa, Sabine, Sander, Sameer, Sorcha, Stefanie, Toms for the nice time, the feedback on papers and bitching about other people. I want to thank the machine translation group for giving me this wonderful opportunity to do a PhD in the first place and helping me grow as a researcher. Thank you Barry, Uli, Rachel, Dominikus, Bonnie, Jozef, Marcin, Sonya, Ana, Maxi, Alham, Christian, Phil, Rico, Faheem, Roman. I thank Hieu for taking me as a student during my undergraduate and helping me with my first steps in C++. I thank Kenneth for the discussions and guidance on high performance optimization, as well as for the countless friendly conversations we've had as I dropped by your office. I give very special thanks to Adam for taking me as a student when he knew so little about me, for his patience with my continuous refusal to learn tikz (or math), for supporting me emotionally and encouraging me through my lows, for insisting that I learn to read and write and understand papers well and for always asking the question: "What does this tell us?" (And the ever present "Can you draw a figure of that (using tikz preferably)?"). I thank Lexi for being a friend to me, for the countless times we've gone for a coffee and laughter and for teaching me how to navigate the complex academic world. I thank Stefan, Ivo, Rumi, Jenny, Esther, Yves, Schuyler (male), Schuyler (female), Vyara, good friends and treasured flatmates, with whom we have shared laughs, home cooked food and woes over all my years in Edinburgh, California and Berlin. I thank Marto, George, Svetla, Irina, Sparkz, Boiko, Nasco, Nicky, Dani, Bats for the many hours online spent playing Dota, for the awesome time in real life, for the seaside holidays, for the friendship that has endured since high school and for

Grants

Table of Contents

Chapter 1

Introduction

Every day, Twitter users write 500 million tweets, Facebook users produce 734 million comments, and Google news serves stories from more than 50 thousand news agencies.[1][2][3][4][5] Three billion people can access this text, but they can each understand only a small fraction of the knowledge it contains, because most of it is in one of the thousands of languages they don't speak. In order to make the world's knowledge accessible to everyone, it must be translated into everyone's language. To do so, translation must be automated, and the demand for this automation is clear: Google translate serves more than 200 million people a day, processing more text than all the world's human translators process in a year.[6] The scale of machine translation requires immense computational cost, hence financial cost. Cost also limits the speed of machine translation research, where the goal is to improve the quality of next generation systems. To produce a single experiment, a machine translation researcher must first train a system, which requires translating more than 1 billion sentences using a neural network.[7] To compare two systems, this process must be repeated twice. To compare two systems on five languages — ten times. To truly improve the quality may take hundreds of such experiments. The high computational cost of machine translation leads to low throughput and is a bottleneck to improving its quality as well as availability.

Conveniently, advancements in computer hardware have kept up with the ever-increasing amount of data available. However the nature of these advancements has

---

[1]http://www.internetlivestats.com/twitter-statistics/

[2]https://www.dsayce.com/social-media/tweets-day/

[3]https://www.quora.com/How-many-Facebook-posts-are-shared-every-day

[4]http://thesocialskinny.com/tag/social-media-statistics/

[5]https://www.theguardian.com/technology/2013/feb/25/1

[6]http://www.cnet.com/news/google-translate-now-serves-200-million-people-daily/

[7]For example, Google (Wu et al., 2016) reported that training their system takes 1 week on 8 GPUs.

changed. More than a decade ago the application's performance would improve by upgrading the CPU to a newer one due to increased clock frequency (Figure 1.1).
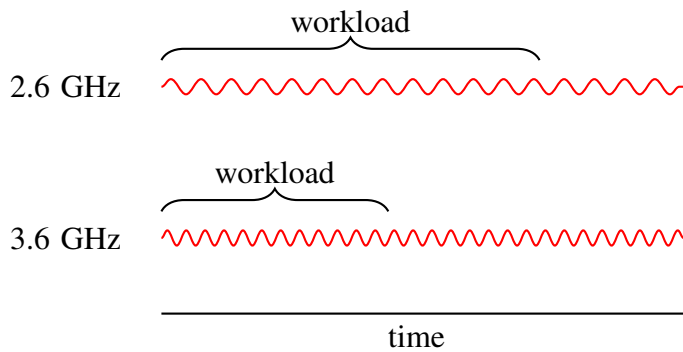
Figure 1.1: Speed improvement with CPU generation upgrades. Each tick-tock (spike) on the graph represents a clock cycle.

New generation hardware fundamentally differs from older generations, because it is highly parallel, with several slower cores per chip instead of one faster one (Figure 1.2). This allows for higher theoretical performance, but only if the application is parallelizeable, which is not always the case. In contrast, a faster clocked CPU would universally yield increased performance regardless of the application design.
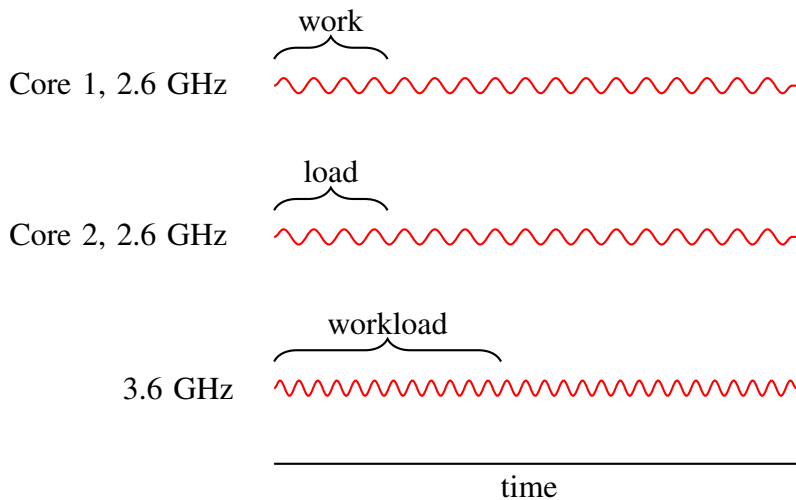
Figure 1.2: Splitting the *workload* in two even parts *work* and *load* over two CPUs results in less wall-clock time necessary for completion of the task, even if the individual CPUs are slower.

In 2010, consumer-grade CPUs had four cores for top-of-the line models, with manufacturers focusing mostly on dual-core high clock solutions. Professional grade CPUs were manufactured with up to eight cores. General purpose graphics processing units (GPGPUs) were just being rolled out, with only a few proof-of-concept applications being written. Mobile phones used slow single core processors. In 2018,

Intel's latest generation of consumer grade hardware offers at least 4 cores and up to 18 cores. AMD's latest generation of consumer grade hardware has gone even more parallel, starting from 6 cores per chip, and up to 32. Mobile hardware is no different, with latest Apple and Android devices featuring dual-cluster multi-core solutions with one high performance multi-core CPU and one power efficient multi-core CPU.[8] In the GPU world the trends are similar with every consecutive generation having more streaming multiprocessors and more cores: From 15 and 2,880 in 2014[9] to 80 and 5,120 in 2018.[10] The trends is to go towards higher core count, rather than faster clock frequency due thermal and physical limits in the CPU design.

Hardware has changed dramatically over the past 8 years, but software has not changed much over the same timeframe. Machine translation toolkits are seldom optimized for parallel hardware because this is a difficult task. Most algorithms are inherently sequential in nature, and as such it is difficult to extract computational parts that can run concurrently (Perrone, 2009). In order for any application to scale (or to even be compatible) with future hardware, it must be redesigned with parallelism in mind (Figure 1.2) and also with the idea that parallelism in the future is likely to increase, while single-core performance is not likely to increase much.

To maximize throughput on modern parallel hardware, the programmer must ensure that all cores are working at all times. But achieving this is difficult; due to slow memory accesses and data dependencies, multi-core CPUs frequently idle:

- **Main memory** is slow to access and could lead to processors idling. If the processor needs to fetch data, ideally the data should be located in the cache, which takes just a few cycles to access. However caches are just a few MB in size, whereas the data structures we work with in machine translation are multiple GB in size and do not fit inside. When main memory needs to be accessed, there would be a considerable amount of time where the processor is unable to do any useful work (Figure 1.3). In order to circumvent this, programmers need to miminize memory accesses and reuse memory as much as possible.

- **Data dependencies** between different CPUs also reduces performance. It is often necessary for one of the processors to access data computed by the other. In this case usually one of the cores needs to wait for the other to finish and

---

[8]Adreno 845/Apple A11
[9]Titan Black
[10]Titan V

in that time no useful work can be done (Figure 1.4). While it is impossible to completely avoid data dependencies, programmers try to minimize them in order to avoid performance degradation.
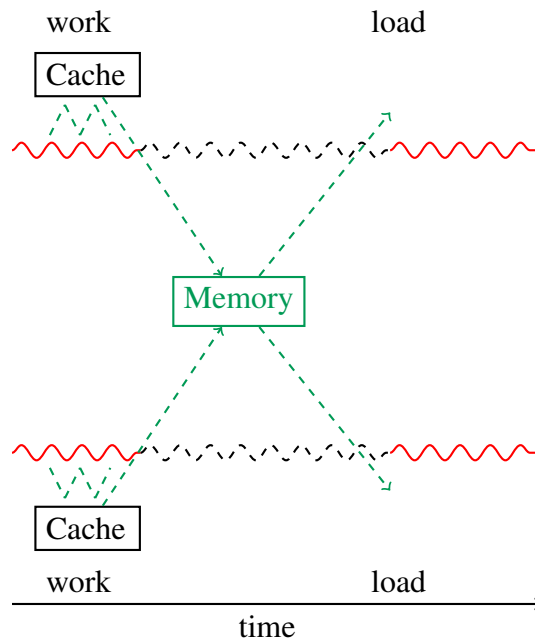
Figure 1.3: Our processors cores need to compute two *workloads*. The processor has access to small amount of fast memory (cache) which allows it to quickly load new data to its registers in order to perform the computation, which serves to complete the first part of the *workload* task (The green dotted line represents memory access). However when the processor needs to go to main memory, in order to fetch the data necessary for the second part of the task *load*, there is going to be a period of inactivity (represented by the black dotted line) where the computation is stalled due to waiting on memory.
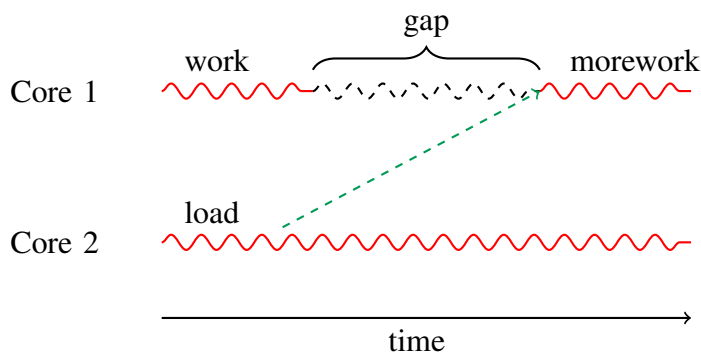
Figure 1.4: Core one and two have split the *workload*, between each other in order to complete it faster. However, before Core 1 can continue to do *morework*, it needs to wait for the results of the Core 2 computation, which creates a processing gap where core 1 idling. Multiple synchronization points could easily kill any theoretical multi-core performance benefits.

This thesis shows that data structures and algorithms can be designed to substantially improve the speed of machine translation systems on modern parallel hardware, by minimizing slow memory accesses and synchronization points.

1.1 Thesis structure

This thesis demonstrates the value of designing new algorithms to minimize memory accesses in three separate use cases in machine translation. In each I identify performance bottlenecks,[11] related to memory access patterns, and implement an algorithmic solution catered towards a contemporary parallel setting. The first part of my work is focused on statistical machine translation and the final part is focused on neural machine translation.

In Chapter 2, I provide brief background on parallel hardware and machine translation.

In Chapter 3, I introduce a novel **phrase table** for statistical machine translation that scales linearly with the increase of core count. Unlike previous implementations, my design addresses the needs of contemporary hardware by focusing on efficient sequential memory accesses, rather than compression. Using this new phrase table, I achieve 10 times faster decoding speed in parallel setting, largely because of the reduced thread contention (Bogoychev and Hoang, 2016; Hoang et al., 2016).

In Chapter 4, I introduce the first GPU **n-gram language model** and I evaluate its performance compared to KenLM (Heafield, 2011), a highly optimized CPU implementation. On the GPU, memory is limited, and every memory access is costly. My language model has a small memory footprint and minimizes the number of memory accesses, which I accomplish using a novel data structure: a BTree-backed trie. This allows for thousands of queries to be executed concurrently on the GPU. The language model achieves 65.5M queries per second in batch mode, which is several times more cost efficient in terms of hardware price than doing the language model queries on CPU (Bogoychev and Lopez, 2016).

In Chapter 5, I show how to improve the speed of **multi-GPU asynchronous training of a recurrent neural machine translation systems** by reducing the frequency of GPU communication. Reducing the communication frequency leads to increased training performance, but worsens the model convergence. I introduce additional methods

---

[11]In this case, and everywhere else in the thesis, the word *performance* is used to mean the speed with which the hardware solves the problem. Where translation quality is concerned I have used *quality*

to tackle the convergence issue and using those I achieve two times faster end-to-end training of a machine translation model at no cost in model quality. Those improvements hold across multiple language pairs and neural network configurations (Bogoychev et al., 2018).

1.2 Contributions

1. I have published three papers as a first author (Bogoychev and Hoang, 2016; Bogoychev and Lopez, 2016; Bogoychev et al., 2018) and collaborated on another two (Hoang et al., 2016; Junczys-Dowmunt et al., 2018).

2. I have implemented a CPU phrase table for statistical machine translation, that is available both for stand alone usage[12] and as the default phrase table in Moses2 (Hoang et al., 2016), a highly optimized statistical machine translation decoder.[13]

3. I have implemented the first GPU based n-gram language model.[14]

4. I have implemented delayed gradient updates for the Marian NMT toolkit (Junczys-Dowmunt et al., 2018), which permits faster training and helps very large models achieve a reasonable mini-batch size.

5. I have worked on the problem of multi-node neural network training on computational clusters with varying connection speed. Together with my collaborators we have implemented a design that minimizes the communication between nodes in order to achieve maximum training speed.

---

[12]https://github.com/XapaJIaMnu/ProbingPT
[13]https://github.com/moses-smt/mosesdecoder/
[14]https://github.com/XapaJIaMnu/gLM

Chapter 2

Parallel programming and Machine Translation

In the 1960s and 1970s, two important trends for the development of computer hardware were observed: Moore's law (Moore, 1965) and Dennard scaling (Dennard et al., 1974). The former establishes that the number of transistors on chips would double every 1.5 years and the latter that the performance per watt would grow exponentially at roughly the same rate. Those laws have influenced tremendously the field of computer science, because they predicted that the performance of processors would increase over the years. Software would run faster just by putting it on a newer processor. Moore's law had to be revised several times, because the growth rate it predicted originally decreased over time, but Dennard scaling started to break down completely around 2005-2007 (Figure 2.1). There are several reasons for that:

1. With the increased voltage and transistor density on a single chip, it became impossible to dissipate the heat quickly enough (Olukotun et al., 1996).

2. With the increased logic circuits and components on a single chip, it became impossible to hide the delay in communication between gates placed physically far away from each other. In order to solve this problem, a greater percentage of the increased transistor count go towards wiring, rather than additional logic gates (Olukotun et al., 1996). A more efficient use of the extra circuits available is to put them to use in an extra processor core. Putting the extra circuits in an additional core allows for overall higher theoretical performance of the new processor, but the burden for achieving it is placed on the programmer.

3. There are physical limits on the delay between logic gates: Signal can't pass

through them faster than the speed of light (McFarland et al., 1995).
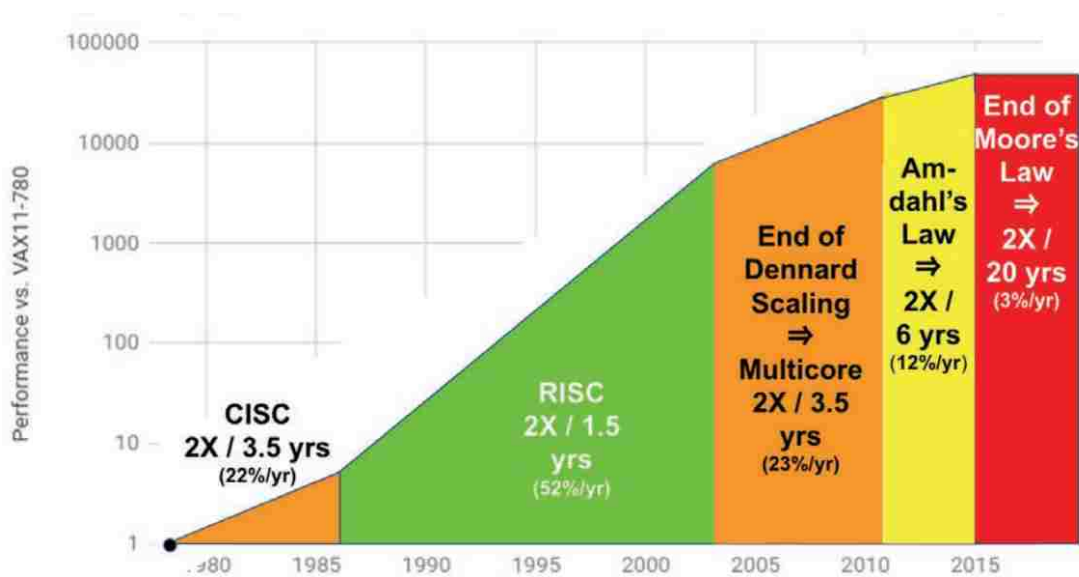
Figure 2.1: Single core CPU performance on a simple Integer benchmark (SPECintCPU) compared to a 1977 VAX 11-780 processor. Single core scaling breaks down completely after 2015. Image source: Hennessy and Patterson (2018)

Furthermore the nature of the workload has changed the past 20 years. Greater processing power and advances in hardware have given birth to the era of big data. In this era, single instruction, multiple data (SIMD) workloads are increasingly common and multi-core processors are better suited for those. This development, together with the aforementioned difficulties in scaling single-core processors, has swayed chipmakers in designing hardware that is better suited to those workloads: multi-core and massively parallel (vector) processors.

2.1   Parallel hardware

Parallel hardware has been conceived for the first time in the mid-1950s and over half a century has become completely integrated even in the most basic consumer level hardware. Parallelism in hardware is found on multiple levels: On a single chip, on a single machine, and on a single network.

2.1.1   On a single chip

Superscalar architecture   Contemporary processors are capable of executing multiple instructions concurrently. There are several ways to take advantage of those capa-

bilities:

- **In software** through out-of-order execution where multiple instructions can be executed in any order as long as this doesn't influence final outcome.

- **In the processor itself** by speculatively executing branches in order to avoid stalls.

- By using **hyperthreads** and letting the OS schedule additional work on the processor. I will revisit the concept of hyperthreads in detail in Chapter 3.

Identifying instructions that can run in parallel is, however, a difficult task and frequently the superscalar capabilities are underutilized.

Symmetric multiprocessors   Symmetric multiprocessors are the most common type of parallel hardware that is used today. They are called symmetric, because as far as the programmer is concerned, all CPUs are equal and have equal access to the main memory. They can be used for any type of parallel operations from multiple instructions, multiple data (MIMD), such as producer-consumer problem to single instruction, multiple data (SIMD), such matrix multiplication.

Massively Parallel hardware   Massively parallel hardware is characterized by having a large number of processors on a single chip, which are usually not completely independent. For example, Nvidia Pascal Titan X is a graphics processing unit (GPU) that contains 3,584 cores that are controlled by 28 streaming multiprocessors, which means at most 28 different instructions can be executed at a time. Another name used for this type of hardware is **vector processor** and they are specifically designed for SIMD workloads. Individual GPU cores are simple, lacking complicated branch predictors or caches, but because of the sheer number of them, collectively they offer much higher theoretical performance than CPUs, given a suitable workload.

Massively parallel architectures share a lot characteristics, because they are designed to tackle SIMD problems, and as such an algorithm that is efficient on one massively parallel architecture is likely to be efficient on another: An algorithm designed for a GPU would map well onto Xeon Phi, a massively parallel CPU board with 57-72 cores manufactured by Intel.

2.1.2   On a single Machine

Most workstation class computers nowadays come with independent CPUs and multiple GPUs. The CPUs are connected through a very fast interconnect and appear to the programmer in the same way a symmetric multiprocessor would. However as the RAM is split between the CPUs (Figure 2.2) accessing memory that was processed by a different physical CPU is more expensive (For precise numbers refer to Table 2.1). The same applies to use multiple GPUs in the computation: While an additional GPU offers theoretically doubled performance, in practise it is difficult to achieve it because of the large cost (Table 2.1) of inter GPU communication. I look specifically at multi-GPU scaling in Chapter 5.

2.1.3   On a single network

Just as we can have multiple computational units within a single machine, we could have multiple machines connected over a network forming a **distributed** computational cluster. Computational clusters allow for potentially thousands of GPUs and CPUs to work together towards a common computation, however programming for them is challenging: When working with distributed hardware memory access to different units is non-uniform (Figure 2.2)—that is to say, memory is organized in a hierarchical manner where the latency of remote memory is much higher than the latency of local memory and the speed of local memory is much higher than the speed of remote memory. As such, algorithms must be designed in a manner that minimizes accesses to the slow remote memory.

2.2   Why parallel programming is hard

There is a subset of programming problems which are labelled *embarrassingly parallel* that exhibit perfect parallelism. They are characterized by lack of communication between cores, which means that no synchronization points are necessary and the cores operate completely independently. An example of perfectly parallel problem is neural network **inference**, where one can generate predictions using the neural network on as many cores as they want in a SIMD manner since the neural network parameters are read only and there is no inter-core communication. Unfortunately, most real world problems require communication in parallel setting and also exhibit difficulties with balancing work evenly between the available computational units, which is notoriously
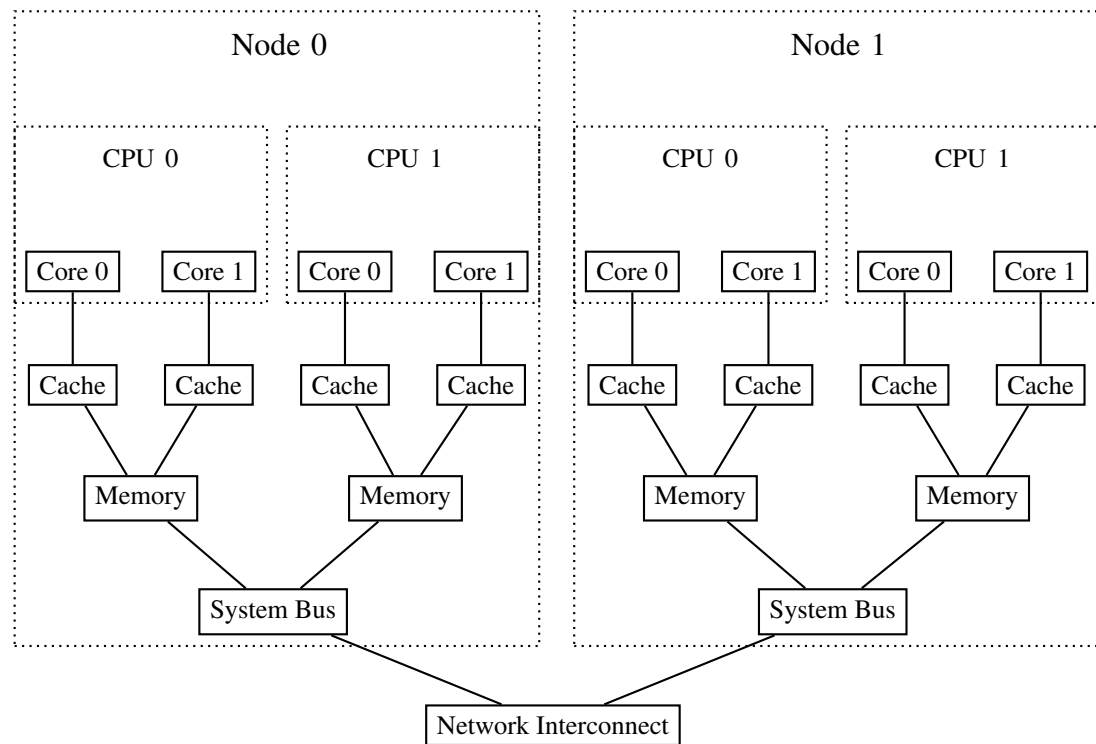
Figure 2.2: Typical memory hierarchy on a computational cluster. Each Node on the cluster has multiple CPUs, each of them has multiple physical cores. Each core has local cache. Each CPU controls a portion of the system memory. Different CPUs communicate to each other's memory through the system bus and communication speed depends on their physical distance, among other factors. In addition to that all nodes are connected through a network and can access each-other's memory resources, but at a much higher latency and lower speed.

difficult (Perrone, 2009).

Parallel programming requires programmers to deal with issues that do not exist in single threaded setting:

Data hazards occur when data written by one thread needs to be read by another. When those threads are scheduled on different cores (or processors) writes need to be propagated through memory in order to be visible to everyone, which requires explicit synchronization or use of atomic instructions, both of which are expensive: Synchronization leads to CPUs idling, and atomic instructions block any sort of instruction-level parallelism, which is very important for nowadays' superscalar processors to achieve peak performance (Schweizer et al., 2015).

Load balancing is the problem of distributing the workload evenly across multiple available computational cores. This part is particularly challenging because a lot of commonly used algorithms are inherently sequential and it is not easy to

extract independent pieces of work that can be done in parallel.

For example, binary search is a widely used search algorithm that is inherently sequential and as such it is impossible to load balance it between multiple processors. Figure 2.3 shows querying a sorted array of numbers for the position of the number **15**. In binary search, at each time step, we take the middle element of the array and we compare it to the number queried. Based on that we discard the half of the array that we know does not contain our number and we repeat the process. Eventually we narrow down the array to two elements and we manage to find our number with just $\log_2(n)$ comparisons and $\log_2(n)$ time, which is much faster than a full scan of the array, however because the input of each step depends on the output of the previous step, parallelization is not possible.

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 | 47 | 53 | 56 | 57 | 60 | 61 | 63 | 66 | 67 | 74 | 78 | 81 |

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 |

| 10 | 12 | 13 | **15** | 17 |

| **15** | 17 |
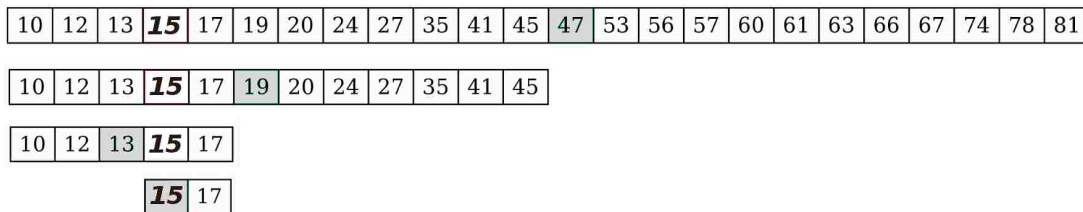
Figure 2.3: Execution of a binary search for key 15. Each row represents a time step and highlights the element compared to the key. Finding key 15 requires four time steps and four comparisons.

On a parallel hardware however we have to change the algorithm to k-ary search in order to take advantage of the architecture. In k-ary search, we essentially split the array in k blocks where k is the number of parallel executing cores and we perform k binary searches in parallel (Figure 2.4). While this solution performs more comparisons in total: $(k-1)\log_k(n)$, they are executed in parallel by $k-1$ devices, resulting in overall runtime of just $\log_k(n)$. As k-ary search maps better to the parallel architecture, it performs the search over the same array in just two timesteps, as opposed to four.

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 | 47 | 53 | 56 | 57 | 60 | 61 | 63 | 66 | 67 | 74 | 78 | 81 |

| **15** | 17 |
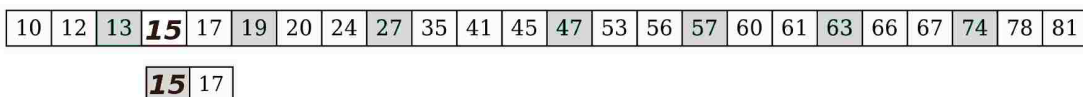
Figure 2.4: Execution of K-ary search with the same input as Figure 4.5, for $K = 8$. The first time step executes seven comparisons in parallel, and the query is recovered in two time steps.

2.2.1 Memory

In this section, I give an overview of memory and why its efficient usage is the key to fast computation.

2.2.1.1 Memory speed

Even in perfectly parallel setting, the programmer must be very careful of the memory usage. Memory speed grows 50% slower than compute speed (Carvalho, 2002) and it often bottlenecks execution time (Patterson et al., 1997). Table 2.1 and Figure 2.5 show approximate time in which CPUs and GPUs need to wait to access a piece of memory. Memory accesses are especially important in parallel and massively parallel setting, because each core or GPU has memory associated with it and accessing remote memory is much slower than accessing local memory (Figure 2.1). Main memory is large and cheap to produce, but accessing it is slow. On the other hand registers and CPU caches much faster, but small and expensive to manufacture (Figure 2.5). In order to avoid expensive trips to main memory, programmers have to carefully design their application so that it minimizes memory accesses and reuses memory where possible. Often, optimizing parallel applications is a matter of reducing memory accesses and where this is not possible, rearranging them in such a manner that one memory access can be used by multiple cores at the same time.
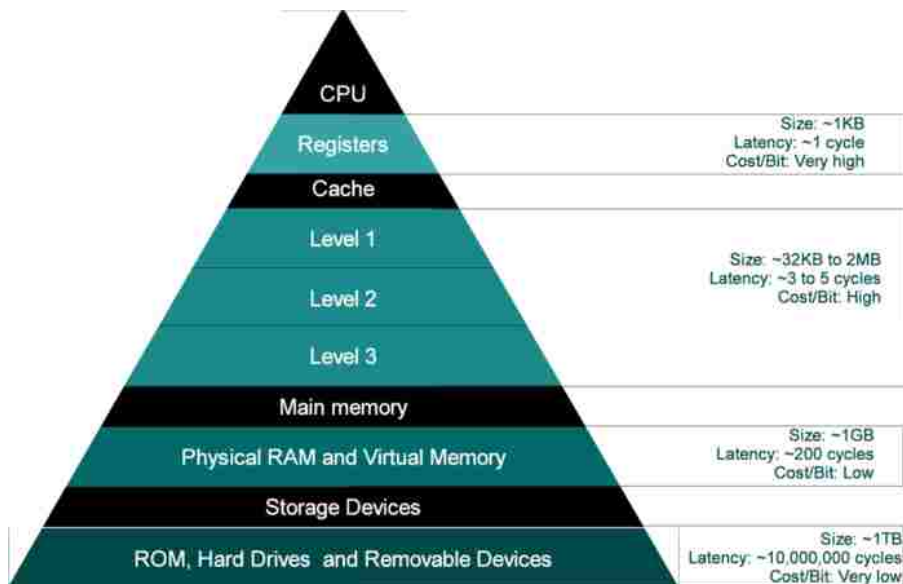
Figure 2.5: Memory hierarchy in the CPU world. Source: Andalam et al. (2013)

CPU		GPU	
Memory type	Latency	Memory type	Latency
Register	0	Register	0
L1 cache	4	Shared	4–8
L2 cache	10	Global GPU	200–800
L3 cache	40	CPU	10K+
Remote L3*	80	Remote GPU	22K+
DRAM	330+		
Remote DRAM	660+		

Table 2.1: Latency (in clock cycles) for accessing different types of CPU and GPU memory. Estimates are adapted from Intel Corporation (2009); NVIDIA Corporation (2015) and also depend on several aspects of hardware configuration. Time for accessing remote GPU devices or DRAM associated with different CPUs is more than twice as slow as accessing local memory.

*Remote L3 refers to accessing cache line shared with a different core that is physically located on the same chip.

2.2.1.2 DRAM bursts and coalesced memory accesses

In order for a programmer to take full advantage of the available memory bandwidth, he/she needs to be mindful of the DRAM burst effect. It is observed that if an application accesses a piece of memory, it is likely that it would also need to access adjacent pieces in the very near future. Because of that main memory is designed to fetch surrounding bytes, together with the requested bits. This effect is known as DRAM burst and its size varies between architectures, but is usually between 32 and 256 bytes. This is the reason why random memory accesses are much much slower than consecutive ones: The consecutive ones can be made essentially for free, as long as they fit within the DRAM burst, whereas random accesses can't make use of all the available memory that has been fetched. In the GPU programming world, where it is usually the case that one thread operates on one unit of data, random memory accesses are particularly harmful, just because there are a lot more of them happening in parallel. In this case it is very important make sure that consecutive threads operate on consecutive pieces of memory, which makes the memory accesses **coalesced**. On the other hand if memory accesses from consecutive threads are not consecutive, the memory access pattern is called **strided**.

2.2.1.3 Data sharing

Data sharing is also problematic: Sharing reads and writes between different threads requires deterministic behaviour which can only be achieved with locking. Locking is expensive because it both stalls some of the cores and involves an expensive context switch to kernel space. For example, in a multi-threaded web crawler where all threads read URLs from a the same queue, every time a thread reads a URL, it needs to be removed from the queue. To ensure that each thread will get a different URL, the queue must be locked before it is modified.

2.3 Machine Translation

Machine translation is the problem of performing an automatic translation from one language to another, for example, the English sentence *Quickly go to the green house!* into Spanish *¡Vete rápidamente a la casa verde!* This is achieved by first training machine learning models on large amount of parallel sentences and then using those models to predict the translations of unseen data.

2.3.1 Statistical Machine Translation

In statistical machine translation, we model the problem as a process of first translating individual words or phrases, and then reordering these translations as shown on example 2.6.
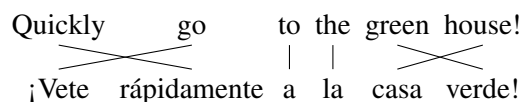
Figure 2.6: English-to-Spanish translation with word-to-word correspondence.

Since there may be many possible translations of a word or phrase, we require a probabilistic model to choose between them – a translation model. The translation model $P(f \mid e)$ provides the probability of a source language phrase f is the translation of the target language phrase e. We combine it with a language model $P(e)$ that gives the likelihood of the sentence e occurring in the target language. Since a phrase table does not contain all possible phrases, the decoder may **reorder** some words in order to attempt to produce a more fluent translation (in practise the decoder may also reorder whole phrases around if it deems it necessary to produce a more fluent output). In

order to find the highest scoring translation $\hat{e}$, we need to perform a search over all possible reorderings $E(f)$ (Equation 2.1).

$$\hat{e} \propto \underbrace{\arg\max_{e \in E(f)}}_{\text{Search}} \underbrace{P(f \mid e)}_{\text{TM}} \times \underbrace{P(e)}_{\text{LM}} \tag{2.1}$$

2.3.1.1   Translation model: $P(f \mid e)$

The **Translation Model** is the $P(f \mid e)$ part of equation 2.1. It provides translations between words of phrases across languages. Languages rarely have one-to-one mappings so we have to deal with a probability distribution over possible translations. For example, the English word *go* could translate into Spanish as *vaya*, *vete*, *va*, *ir* or a myriad of other present tense conjugations due to the more rich tense system in Spanish. The job of the translation model is to provide us with the probability of each possible mapping of the words or phrases across the languages in concern.

Working through my small example, let us imagine how a simplified phrase table would look just for the verb "go" and the phrase "go to the":

English	Spanish	Probability
go	ir	30%
go	va	30%
go	vaya	10%
go	ve	10%
go	vete	10%
go	va	10%
go to the	vaya a la	30%
go to the	vaya al	30%
go to the	vas a la	15%
go to the	vas al	15%
go to the	vete al	5%
go to the	vete a la	5%

Table 2.2: An extract from a Spanish-English phrase-table for the word "go" and the phrase "go to the". In this case English is the source language f and Spanish is the target language e.

This is the phrase table ambiguity that occurs just for quarter of the words in the example sentence. Phrase tables' size could reach hundreds of gigabytes in size (Lopez,

2007, 2008; Germann, 2014; Bogoychev and Hoang, 2016) and its query speed is crucial in order to achieve fast decoding speed. In Chapter 3, I show how my improved phrase table design leads to order-of-magnitude increase in decoding speed in parallel setting.

2.3.1.2 Language model: $P(e)$

The **Language Model** is the $P(e)$ part of equation 2.1. The translation model has provided the possible translations of the phrases, but there is no way of knowing which ones would be the most appropriate. The language model is a statistical representation of a language which captures likely sequences. During translation we can use the language model to choose the most likely permutations from the translation model that constitute fluent speech in the target language. *N*-gram language models with a smoothing strategy (Chen and Goodman, 1999) are most commonly used in statistical machine translation. Here *N* refers to the maximum context window size of the language model.

If we don't have the full sentence in the phrase table, we build our translation out of smaller phrases that we have seen already in our training data. However as languages do not have a monotonic or one-to-one mappings, a single source word is often translated into multiple target words depending on the context. In order to disambiguate, we require a language model to tell us how likely the sequence we have produced is and to potentially reorder words in some phrases in order to achieve a more natural translation. In my example, if my phrase table does not contain the phrase *to the green house ↔ a la casa verde*, we would have to use unigram translations. In this case (and assuming all other parts of the sentence are built correctly) we would build the following translation:

This sentence is not grammatical, due to the words *casa* and *verde* appearing in the wrong order and would be given a low score by the language model. Our translation system would attempt to produce a better candidate sentence by trying out different permutations of the target words to be scored by the language model as shown on Figure 2.7.

This process would be repeated with every phrase used to construct the sentence,

Quickly   go   to   the green house!

¡Vete   rápidamente verde   a   la   casa!

Quickly   go   to   the   green house!

¡Vete   rápidamente a   verde   la   casa!

Quickly   go   to the green house!

¡Vete   rápidamente a   la   casa   verde!
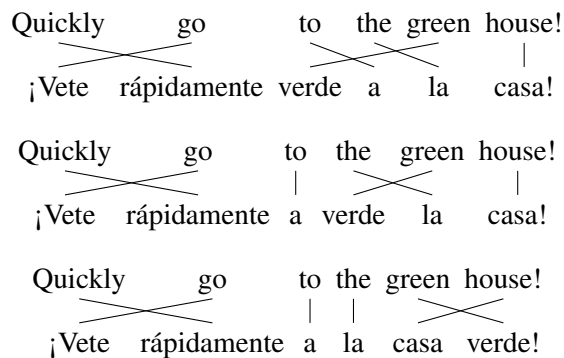
Figure 2.7: Example permutations of the Spanish translation of *to the green house*.

thus thousands of similar permutations need to be produced. In practice there is a limit
on the distance any given word can move in order to reduce the search space.

On average, more than 700,000 language model queries are done per sentence de-
coded (Heafield, 2013) which takes up more than half of the wall clock time (Green
et al., 2014). In statistical machine translation, the language model is the most im-
portant component for achieving fluent target side sentences and thus researchers and
industry use the largest model they have (or the largest model their system can fit in
memory). Google are known to use an *n*-gram language model with 2 billion tokens
and more than 300 billion *n*-grams (Brants et al., 2007). In Chapter 4, I show how we
can use GPUs to accelerate *n*-gram language model queries.

2.3.1.3 Search: $\hat{e} = \arg\max_{e \in E(f)}$

Search is the $\hat{e} = \arg\max_{e \in E(f)}$ part of equation 2.1. For any given sentence all possible
permutations of the translation model outputs are scored using the language model
and the translation model. The sequence with the highest probability is the chosen
translation. The search space increases exponentially with sentence length, because of
the one-to-many translation mapping and the possible phrase permutations. Searching
exactly for the best translation involves producing hundreds of thousands translations
that are evaluated by the language model (Koehn, 2010). In order to make it tractable
various pruning methods are used (Tillmann and Ney, 2003; Chiang, 2007), which
find an approximate solution. Unfortunately, while those approximation techniques
significantly reduce the computational cost, they are also inherently sequential and
hard to parallelise. Because this thesis does not focus on search, I do not go into
further detail here.

2.3.2 Neural Machine translation

Neural machine translation does not employ any of the statistical methods from statistical machine translation. Instead it employs a neural network to directly model $P(e \mid f)$ by conditioning every single target word e_i on all previously produced target words $e_1, ..., e_{i-1}$ and the full input sequence f:

$$P(e \mid f) = \prod_{i=1}^{|f|} P(e_i \mid e_0, ..., e_{i-1}, f_1, ..., f_{|f|})$$

The most popular architecture, designed by Bahdanau et al. (2014), is a bidirectional long-short term memory (bi-LSTM) recurrent neural network (RNN) with an encoder, decoder and attention mechanism. The details of these models are not crucial to understanding this thesis, so I do not describe them in detail here. Neural network parameter optimisation, or **training**, is performed using stochastic gradient descent, most commonly using **mini-batches**, tiny subsets of the training data: The training data is split into many mini-batches and then for each of them we compute a forward pass through the network, to get the predictions and the error, and a backward pass in order to get the gradient and update the parameters. As such, neural network training is roughly three times more expensive per sentence compared to inference, therefore parallelising training across multiple GPUs is important in order to reduce training times. Chapter 5 explains in details neural network training.

Neural machine translation training is very computationally expensive due to the large number of model parameters – over 200 million that form up the weight matrices[1], as well as the large number of sentences that are passed through the network during training – over 100 million.[2] Training those systems takes weeks on multiple GPUs (Wu et al., 2016), therefore training speed is a limiting factor for neural machine translation: The large amount of time it takes to train the model to convergence limits the amount of experiments that are able to be completed, the training data that is able to be used and the size of the neural network.

Goyal et al. (2017) have shown that neural network training can be substantially sped up by increasing the mini-batch size and using large amount of GPUs (256). They train ImageNet, a large scale image classification task by Deng et al. (2009), in just one hour, however the approach they take does not translate directly to the neural machine

---

[1]Measurement taken from the Haddow et al. (2018) models.

[2]The corpora are actually much smaller than that, but they are passed through the neural network multiple times

translation setting, because of the different type of neural network (convolutional vs recurrent) and the different type of data. In Chapter 5, I show how we can substantially increase the speed of neural machine translation training, by applying some of the methods by Goyal et al. (2017) and my own original work.

2.4   Conclusion

In this, chapter I have discussed the physical limitations that have ended single core scaling and thus driven hardware manufacturers towards parallel and massively parallel devices. Multi-core and massively parallel devices work best with small data structures in compute heavy workloads that require little to no synchronization, which is at odds with the workloads typical for machine translation. Statistical machine translation works with large data structures that put a lot of strain on the memory and multi-device neural machine translation training requires frequent synchronization points. In the next three chapters, I show how we can extract better performance from modern parallel and massively parallel hardware in both statistical and neural machine translation setting.

Chapter 3

Phrase table for for symmetric multiprocessors

Phrase tables are the most basic component of a statistical machine translation decoder, containing the parallel phrases necessary to perform phrase-based machine translation. A phrase table is a dictionary that maps source phrases to target phrases. It is used in statistical machine translation to produce a list of possible target phrase translations, given a source phrase. Phrase tables typically reach hundreds of gigabytes in size, and Lopez (2008) describes phrase tables that reach half of terabyte in size. A decade ago it was prohibitively expensive for a phrase table of this size to reside in memory, even if hardware supported it: a gigabyte of RAM cost about a 100 USD in 2006, compared to 5 USD in 2018. Because of that for a long time machine translation was considered a big data problem and the engineering efforts were focused on reducing the memory footprint. Phrase tables were implemented using a number of different data structures which all tackled the memory usage problem in a different manner:

- Zens and Ney (2007) developed a prefix tree (commonly known as a trie; Fredkin, 1960) based phrase table that drastically reduces the memory usage. The phrase table was designed with the idea that the trie is to be an external data structure residing on disk, as it is too big to be loaded in memory in its entirety and subsections of it are only loaded on demand.

- Callison-Burch et al. (2005); Zhang and Vogel (2005); Lopez (2007); Germann (2015) developed suffix array based phrase tables, which work directly with the parallel corpora, extracting target phrases on the fly, in order to enable easier addition of new data and keep memory usage low. This is great in terms of flex-

ibility and memory usage, but comes at the cost of speed, as the suffix arrays are searched using binary search, which requires $log_2(T)$ random memory accesses where T is the number of tokens in the parallel corpora, which is detrimental to performance.

- Junczys-Dowmunt (2012) developed another phrase table that focuses on compression and small memory footprint. It uses phrasal rank encoding (Figure 3.1), which can be viewed as a form of phrase level byte pair encoding (Gage, 1994). The method recursively encodes bigger strings as a composition of several smaller ones until only small units remain. Source phrases that do not expand further point to a list of target phrases to which they correspond, ranked by probability. Figure 3.1 shows shows the retrieval process for the source phrase *Maria no daba una bofetada a la bruja verde* from the phrase table.

  - Querying the initial source phrase returns two triplets of numbers, alongside plain text *green witch*. This means that for reconstructing the target phrase we need to query two subphrases, defined by those triplets, as well as concatenate the resulting target phrases with *green witch*, which is stored as plain text here because it only occurs once in the corpora.

  - The span *{0,4,0}* encodes the following information: The difference of indexes between the start the source phrase and the target subphrase, the distance from the right subphrase to the end of the encoded phrase and the rank of the target phrase among all target phrases that correspond to this context. Using this information we identify the next subphrase that needs to be queried, *Maria no daba una bofetada* and obtain a corresponding pair of triplets.

  - We follow the next triplet to first get *Maria no* and later two separate unigram queries of *Maria* and *no*.

  - *Maria* has only one translation so it has rank 0 and we obtain the corresponding target phrase *Maria*, however *no* has multiple possible translations. This is why the triplet corresponding to *no*, *{0,0,2}* identifies that we are interested in the translation ranked 2, *did not*.

  - The procedure is repeated until all subphrases reconstructed. The cost of querying this phrase table during decoding is potentially exponential in terms of sentence length as phrase based decoding proceeds left-to-right
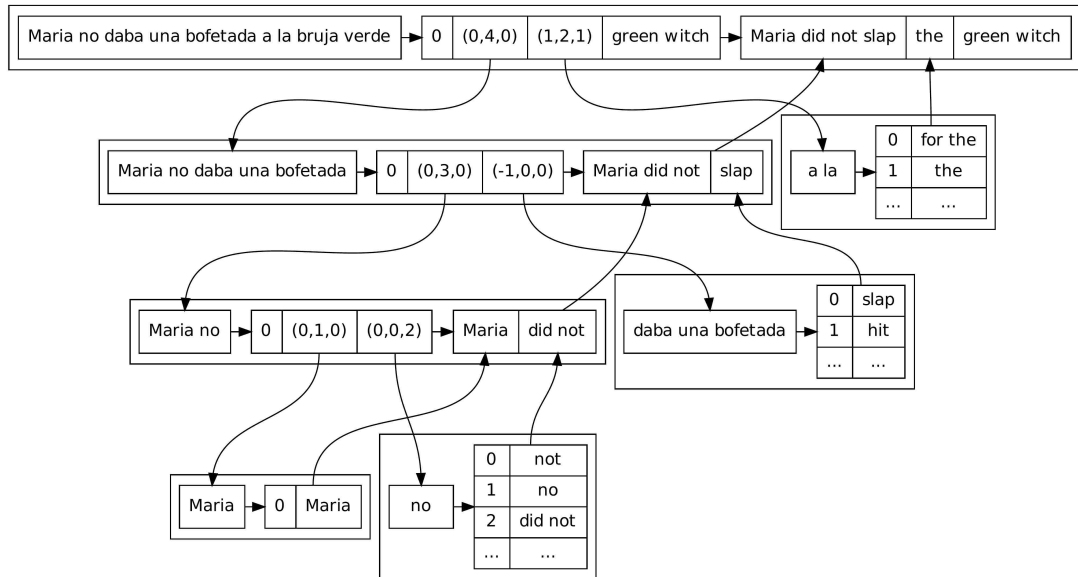
Figure 3.1: Phrasal rank encoding. Figure taken from Junczys-Dowmunt (2012)

.

and will query all subphrases of a sentence. Extensive caching is used in order to avoid repeated subphrase expansions.

In order to further reduce memory footprint, minimum perfect hashing (Nick Cercone, 1983) is used to hash phrases to their expansions and on top of that bit-aligned Huffman encoding is used to further compress the phrases.

The rest of the chapter is structured as follows: First, I describe the problems with the current implementations. Then, I describe my own implementation that addresses the aforementioned shortcomings and finally I evaluate the performance of my novel phrase table against existing implementations.

3.1   Problems with current implementations

RAM prices have dropped 20 times over the past decade and high performance server machines have hundreds of gigabytes of memory. For those machines it is no longer needed to sacrifice query performance in favour of compression techniques such as the ones described above. The implementation of Junczys-Dowmunt (2012), which was the state of the art phrase table in terms of both speed and space usage in 2012 is slow on modern parallel hardware because:

Many random memory accesses   Phrasal rank encoding, together with Huffman encoding make the implementation of Junczys-Dowmunt (2012) extremely space efficient, but at the cost of speed, because of the potentially exponential number of random memory accesses necessary to reconstruct a single phrase as shown on Figure 3.1. As discussed earlier, large number of random memory accesses make poor use of the DRAM burst, which results in very poor memory bandwidth.

Caching is slow in multi-threaded setting   To make queries faster, Junczys-Dowmunt (2012) uses extensive caching. The cache is implemented as a dictionary of source phrases and the corresponding target phrases. To conserve memory all threads used during decoding share the same cache, so the cache needs to be locked when updated. Therefore, when using more than 8 threads during decoding, performance goes down due to the cost of locking.

3.2   ProbingPT

I have designed a new phrase table called **ProbingPT** based on linear probing hash table (Peterson, 1957) for storage and lock-free querying, in order to deliver high performance in modern use cases where memory is not limited. My goal in design was to eliminate the necessity for cache by using data structures that minimize random memory accesses.

Linear probing hash table H is a data structure, based on an array A, that maps a key k to a corresponding value v. A hash function $h : k \to \mathbb{N}$ determines the memory location where v is going to be stored inside the hash table. If several different k point to the same memory location, there will a hash **collision** that needs to be resolved. In order to identify whether a key collision has happened or if we are trying to insert a duplicate entry, we store a fingerprint, the result of a *different* hash function $h' : k \to \mathbb{N}$ together with v. In a linear probing hash table, linear probing refers to the collision resolution strategy of the hash table, where if k points to a memory location in A that is already occupied by a different key, the next available memory location is used instead. This is very efficient, because those memory locations would already be loaded from main memory due to the DRAM burst effect.

When a linear probing hash table is queried we check if k exists at its corresponding memory location. If the memory location is unoccupied, then we can conclude that k and its corresponding v are not found in the table. If the memory location is occupied

we verify that the *k* queried is the same as the one found at the memory location and not a product of collision. If there is a mismatch, we proceed to linearly traverse the phrase table until either we encounter a match of our *k* in which case we return the corresponding *v*, or we reach an empty space or the end of the hash table. The latter two cases mean that *k* and its corresponding *v* are not in the hash table.

I use an existing linear probing hash table implementation (Heafield, 2011). Linear probing hash provides $O(1)$ search time, has a very small overhead per entry stored and is shown to be very fast in practice (Peterson, 1957).

The phrase table consists of three arrays: The first *H* contains a probing hash table that maps hashes of source phrases to a memory location in the payloads array *P*, which contains phrase probabilities, word alignments key, and optionally lexical reordering score, a common feature used in statistical machine translation. The third array is another hash table *A* that contains word alignments. Since there are about a 1000 unique word alignments for the hundreds of thousands of phrase pairs I store them in a separate hash table and in the payloads array I store an extra key for each target phrase that points to its corresponding alignment in the hash table. Hashes of the source phrases are used as keys. The query algorithm goes as follows:

1. The translation of a source phrase is requested from the decoder.

2. The source phrase is hashed to produce a key *k* for which *H* is queried. If it doesn't exist, "NOT FOUND" is returned to the decoder. This is the first random memory access that the phrase table makes.

3. If it does exist, the hash table returns the memory location within *P*. This memory location is where all target phrases, corresponding to this source phrase are stored.

4. The first few bytes at the memory location that was previously identified contain information about the number of target phrases and the number of bytes they occupy. This is the second random memory read. Since everything is laid out consecutively in *A*, this DRAM burst also loads the memory for the target phrases that are about to be extracted.

5. Then, for each target phrase, the phrase table extracts the scores, the alignments and the tokens and returns them in separate arrays so they are easily used by the decoder. All of those elements with the exception of the alignment scores are laid out consecutively in the payloads array. The alignment scores are extracted

from *A*. For each target phrase there is potentially one additional memory access to *A* for the alignment scores.

My new phrase table phrase table address the shortcomings of previous implementations:

Few random memory accesses I have reduced the number of random memory accesses from potentially exponential in terms of phrase length, to just two: One for the large probing hash table and one for the payloads array. My phrase table also also has one additional memory access per target phrase, because it needs to fetch the alignment information from a separate hash table. That hash table, however, is small enough (20-30 KB) to fit in the L1 or L2 cache of any modern processor, which makes accessing it quite cheap: If the memory location accessed is already in cache, the cost of random memory access is not at all high: just 4-10 CPU cycles, compared to 300+ for DRAM.

Caching is now redundant The phrase retrieval procedure of my phrase table is simple and doesn't involve large number of memory accesses, which makes caches unnecessary. This makes the phrase table multi-threading friendly, because there is no need for locking or synchronization of different threads.

3.2.1 Integrated lexical reordering

Lexical reordering scores, associated with every single source-target phrase pair, are frequently used in Moses (Koehn et al., 2007) as a feature function, but they are stored separately from the phrase table inside a reordering table. Since those scores are associated with every single phrase pair, it is possible to attach those scores to the phrase table's entries. Doing so reduces both memory usage and memory accesses: We no longer require a separate hash table for the source/target phrases that the lexical reordering scores are associated with, as I reuse the key from my phrase table. Extracting lexical reordering scores no longer incurs the penalty of additional memory accesses, as querying is tied to the phrase table query and all related scores would be fetched with the same DRAM burst, because they are stored consecutively. To my knowledge, this is the first phrase table implementation that incorporates lexical reordering table.

The phrase table is part of upstream Moses[1] but it can also be used standalone.[2]

---

[1]https://github.com/moses-smt/mosesdecoder
[2]https://github.com/XapaJIaMnu/ProbingPT

3.3 Experimental setup

For the performance evaluation I used French-English model trained on 2 million EU-ROPARL sentences. I used a KenLM (Heafield, 2011) language model and cube pruning algorithm (Chiang, 2007) with a pop-limit of 400. I time the end-to-end translation of 200,000 sentences from the training set. All experiments were performed on a machine with two Xeon E5-2680 processors clocked at 2.7 Ghz with total of 16 cores and 16 hyperthreads and 290 GB of RAM.

Hyperthreads are Intel's way of implementing simultaneous multi-threading, a technique that improves the efficiency of modern superscalar processors. Superscalar processors can execute more than one instruction in parallel, provided such instructions are scheduled. Hyperthreads expose additional CPU cores to the OS, which are indistinguishable from actual CPU cores. A 16 core, 16 hyperthread processor would appear to the OS as a 32 core processor. This allows for the OS scheduler and applications to schedule more instructions to the processor which ensures that there would be enough instructions available to take advantage of the superscalar architecture. It is important to note that hyperthreads do not always increase performance. In scenarios where there is high level of lock contention, the introduction of additional threads in the workload will make the performance go down.

In all of my figures "32 cores" means 16 cores and 16 hyperthreads.

3.3.1 Decoders

I use two different decoders for my experiments: The widely used Moses machine translation decoder and *Moses2*, a redesign of Moses focusing on speed (Hoang et al., 2016). I benchmark using Moses to show the speedup my implementation offers as a drop-in replacement to existing phrase tables in the widely used decoder. Unfortunately, Moses has major multi-threading problems, due to its extensive usage of **std::locale**, which invokes a global lock. As such, Moses is a poor framework for measuring the multi-threading performance of my phrase table, as the underlying multi-threading already cause high lock contention at high thread count. I used the highly optimized *Moses2* to show the speed my phrase table can achieve when it is running on a decoder optimized for multi-threading. Furthermore the *Moses* API made it difficult to include integrated lexical reordering and it is only a feature in *Moses2*.

3.3.2 PhraseTables

In my experiments I compare ProbingPT against CompactPT (Junczys-Dowmunt, 2012). There are currently two other phrase tables: PhraseDictionaryOnDisk, a multi-threading enabled implementation of the Zens and Ney (2007) phrase table and PhraseDictionaryMemory, an in-memory phrase implementation of the work of Zens and Ney (2007) using STL data structures. Junczys-Dowmunt (2012) has shown that CompactPT is faster than both implementations of the Zens and Ney (2007) in all settings. Based on this I focus on comparing ProbingPT against CompactPT.

ProbingPT and CompactPT produced nearly identical translations under the same decoder. In my tests 3 out of 200,000 sentences slightly differ in their translation. This is expected according to Junczys-Dowmunt (2012) because CompactPT may reconstruct a phrase erroneously in a few edge cases.[3] Since my phrase table produces the same translations as both implementations of Zens and Ney (2007), I conclude my implementation is correct and can be used as drop-in replacement for CompactPT.

3.4   Results

Phrase table	Size
ProbingPT	5.8 GB
ProbingPT + Reordering (RO)	8.2 GB
CompactPT	1.3 GB
CompactPT RO	0.6 GB

Table 3.1: Phrase table sizes

CompactPT which is designed to minimize model size has naturally lower model size compared to ProbingPT. However the extra RAM used is only 2% of the 290 GBs available on my test system which is insignificant.

Figure 3.2 shows performance comparison of two systems with CompactPT based reordering tables that differ in the phrase table used. The best performing ProbingPT system here delivers about 30% better performance compared to the corresponding CompactPT system. We see that the CompactPT system doesn't improve its perfor-

---

[3]Personal correspondence
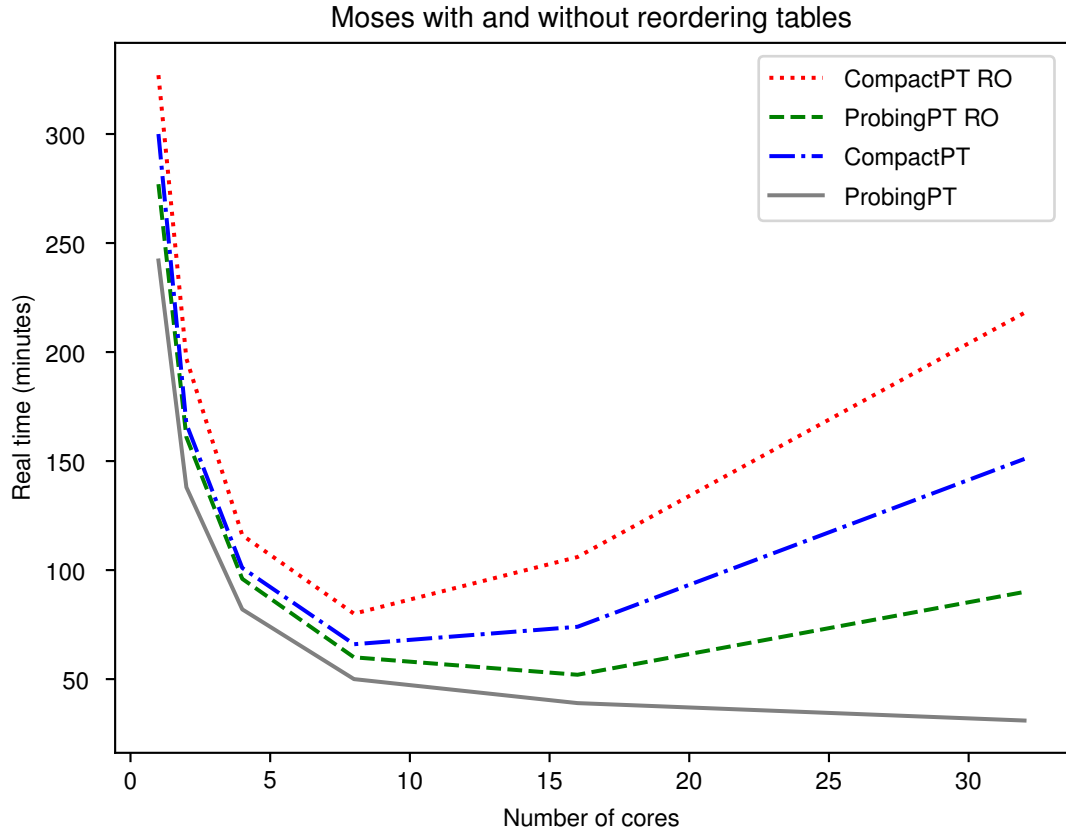
Figure 3.2: A comparison of CompactPT and ProbingPT's time to translate the dataset with different core counts. The ProbingPT systems are better than the CompactPT systems, which stop scaling at 8 threads. ProbingPT scales up to 16 threads when a CompactPT based reordering table is also used, but when it is removed, the ProbingPT system continues scaling.

mance when using more than 8 threads, but the ProbingPT one continues to scale further until it starts using hyperthreads.

I hypothesized that the performance of the ProbingPT system on Figure 3.2 is hampered by the inclusion of CompactPT based reordering. Moses doesn't support ProbingPT based reordering and in order to measure the head-to-head performance of the two phrase tables, I also included in the figure the same test using two systems that do not use reordering tables. We can see that ProbingPT model without reordering table consistently outperforms the corresponding CompactPT by 10-20% at lower thread count but the difference grows as much as 5 times in favour of ProbingPT at the maximum available thread count on the system. Comparing the best performance achieved from both system, ProbingPT is capable of delivering twice the performance of CompactPT. It is important to note that ProbingPT's performance always increases with the

Moses with and without reordering tables, system time
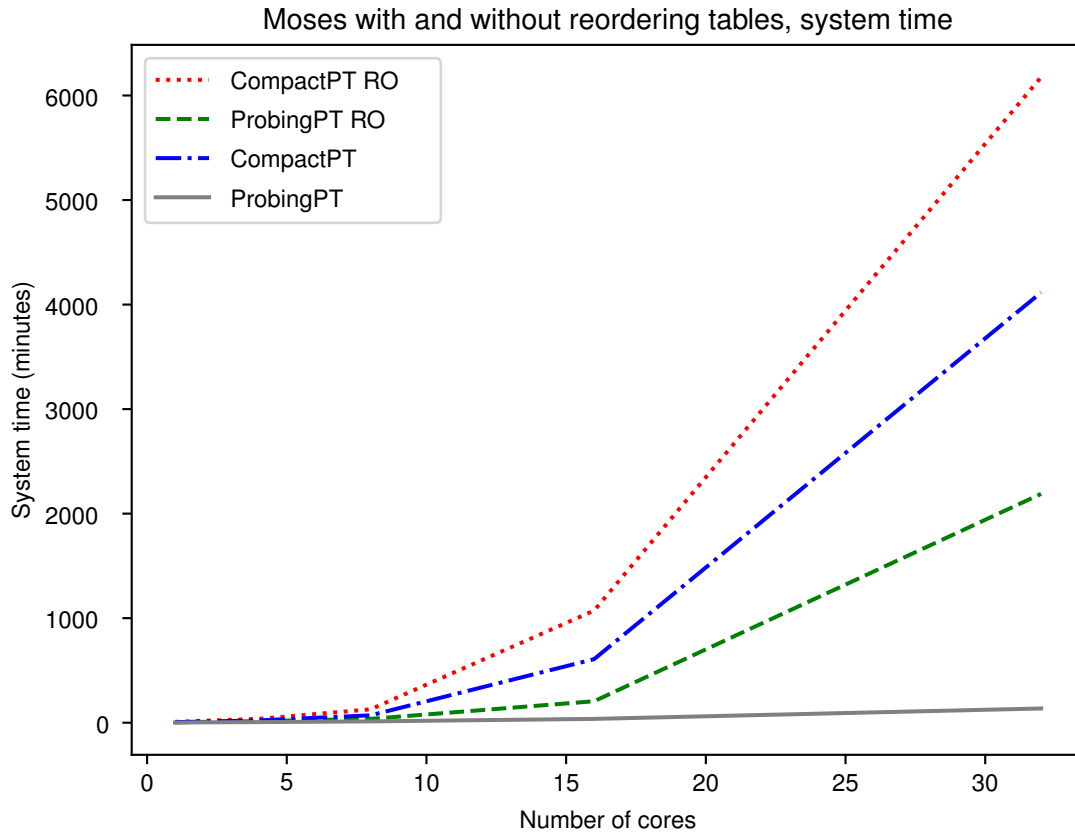
Figure 3.3: System time comparison of the systems on Figure 3.2. System time is larger in terms of absolute value compared to real time, because it is summed across all threads.

increase of the thread count, whereas CompactPT's performance doesn't improve past 8 threads. We can also see that the ProbingPT based system can even take advantage of hyperthreads, which is not possible with any system that uses CompactPT based table (Figure 3.2). Removing the reordering table from the CompactPT system has a much smaller effect than removing it from the ProbingPT system. This shows that the lexical reordering table only slows down the decoder because it is implemented in a inefficient manner. I can conclude that Moses can achieve faster translation times on highly parallel systems by using ProbingPT.

3.4.1   Why is CompactPT slower?

In the single-threaded case it is likely that CompactPT's many random memory accesses cause it to be slower than ProbingPT, because consecutive memory accesses are much faster due to the DRAM burst effect. When the thread count grows, the

performance gap between CompactPT and ProbingPT widens, because of the locking that goes on in the former's cache. This can be seen from Figure 3.3 which shows the system time used in the experiment on Figure 3.2. System time shows how much time a process has spent inside kernel routines, which includes locking and memory allocations and IO operations. We want to avoid spending time in kernel routines as much as possible, because context switching to kernel space is expensive and runtime spent in there is wasteful in terms of computational resources. The system time used in the CompactPT systems grows linearly until 8 threads and then the growth rate starts increasing at a faster rate widening the gap with ProbingPT. This is also the reason why CompactPT's performance severely degrades when using hyperthreads: the extra threads just spend more time waiting on locks, rather than doing useful work. The ProbingPT system without reordering table on the other hand increases its usage of system time at a linear rate even when using hyperthreads. In the worst case, when using 32 threads, ProbingPT spends 16% of its runtime in kernel routines, whereas CompactPT spends 84% of its runtime there at the same thread count. I can conclude that design of ProbingPT, which focuses on memory throughput rather than compactness, scales very well with the increase of number of threads and is suitable for use in modern translation systems running on contemporary hardware.

3.4.2 Order of magnitude performance improvements with integrated reordering table

As integrated lexical reordering is only available in *Moses2* I conducted an experiment where I compare systems using CompactPT based reordering and ProbingPT integrated reordering (Figure 3.4). The ProbingPT based system is able to translate all sentences in my test set in only 4 minutes, whereas the the system using CompactPT reordering system took 39 minutes. I also observed limited scaling when using CompactPT based reordering: the best performance was achieved at 8 threads. I observe that using lexical reordering table has negligible impact on performance if it is used within ProbingPT. I am not entirely certain which factor contributed more to the increased performance: having a reordering table based on the faster ProbingPT or the reduced IO and computational resources that the integrated reordering table requires. As currently there is no standalone ProbingPT based reordering table I can not say for sure. Nevertheless I achieve 10x speedup by using my novel reordering table within *Moses2* (Figure 3.5).
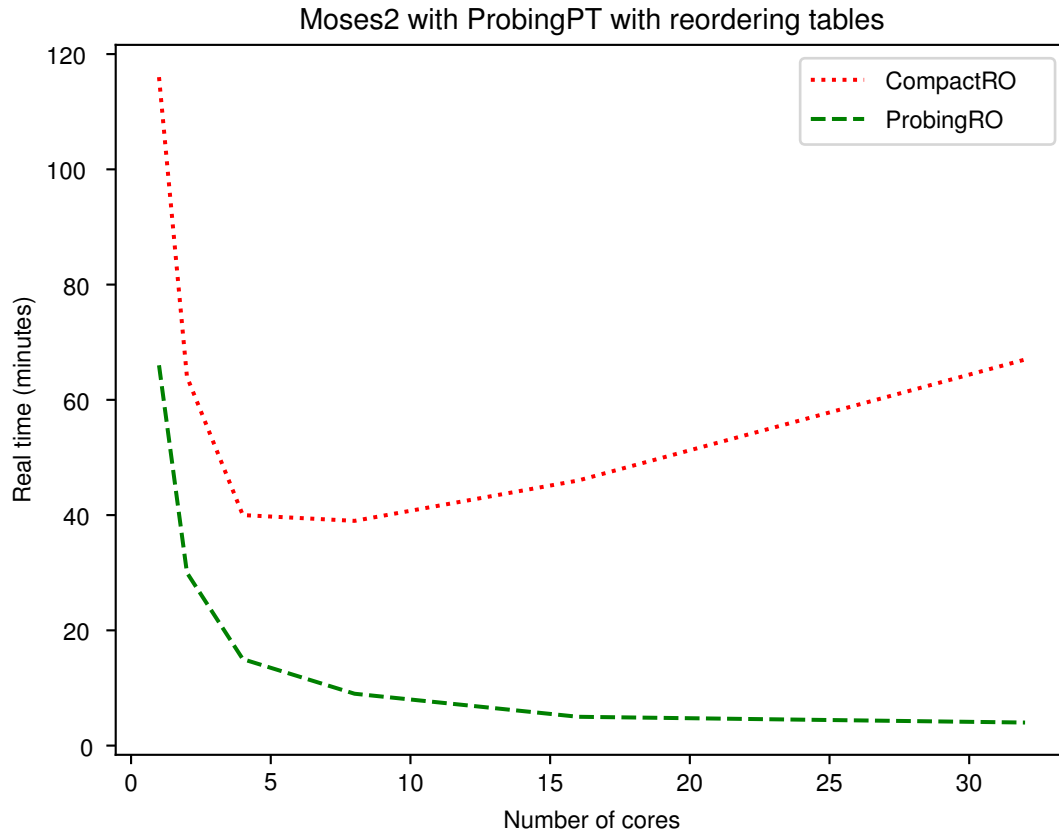
Figure 3.4: *Moses2* comparison between ProbingPT integrated reordering and CompactPT based reordering. Both systems use ProbingPT as a phrase table.

3.5   Evaluation

3.5.1   Profiling the code

I was very surprised of the speedup my phrase table offered, particularly in *Moses2*, because in phrase-based decoding, the number of phrase table queries increases linearly with the length of the sentence. They constitute a tiny fraction of the number of language model queries, which are about 1 million per sentence (Heafield, 2013). To get a clearer understanding of this, I used Google's profiler.[4] I profiled the pair of systems, displayed on Figure 3.4. In the system which has ProbingPT based reordering, the language model is responsible for about 40% of the decoding runtime, compared with only 1% in the *Moses2* system with CompactPT based reordering. In the latter system the runtime is dominated by CompactPT search and **std::locale** locking due to

---

[4]https://github.com/gperftools/gperftools
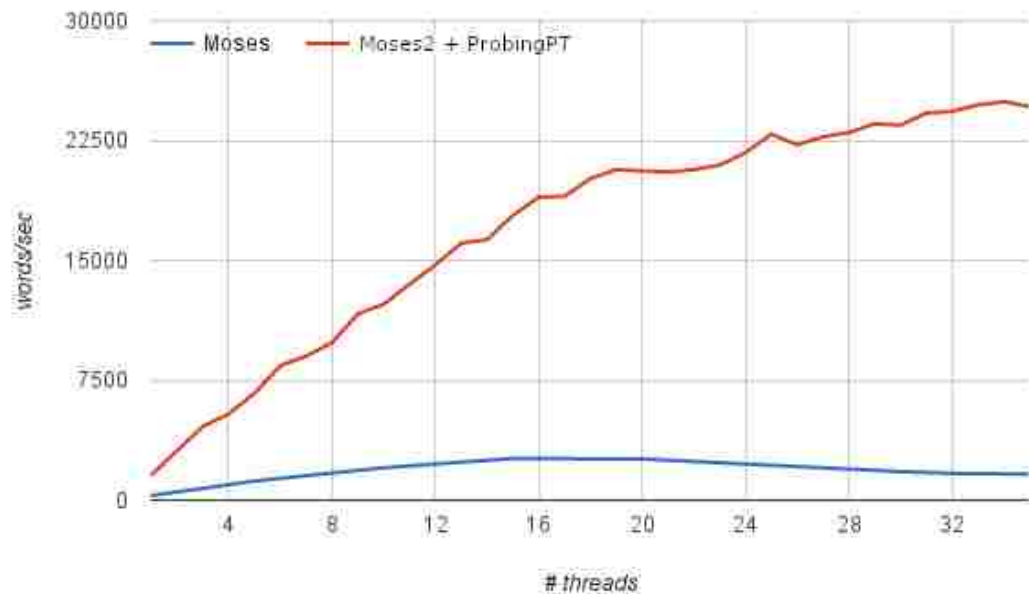
Figure 3.5: Comparison between *Moses* and *Moses2* with ProbingPT in terms of words-per-second decoding performance. Image source: Hoang et al. (2016)

the phrase table using string operations during its search.

In *Moses* the difference between using ProbingPT and CompactPT is not so apparent, before we go to higher thread count, because the decoder itself is very slow and hides the phrase table inefficiencies. It is clear that even though the phrase table queries are a small part of the full decoding process, they are enough to slow it down 10 times if no other bottlenecks exist. Using ProbingPT for both the phrase table and the reordering model makes for a compelling combination.

3.6   Conclusion

I conducted a case study of phrase table performance in a statistical machine translation decoder and identified performance shortcomings in existing phrase table implementations: Modern hardware has abundance of memory, but memory speed is still much lower than the CPU's computational power. Therefore in order to take full advantage of the modern parallel hardware we need to minimize memory accesses and remove synchronization points. Having those in mind I designed a new lock free phrase table, that maximizes memory bandwidth and achieved 10x performance improvement over existing implementations. My phrase table is the central piece of the *Moses2* decoder

and is available as a standalone software that can be used by other researchers.

Chapter 4

N-gram language models for massively parallel devices

N-gram language models are ubiquitous in speech and language processing applications such as machine translation, speech recognition, optical character recognition, and predictive text. Because they operate over large vocabularies, they are often a computational bottleneck. For example, in machine translation, Heafield (2013) estimates that decoding a single sentence requires a million language model queries, and Green et al. (2014) estimate that this accounts for more than 50% of decoding CPU time.

In order to achieve faster decoding time, I attempt to speed up the slowest component by turning to massively parallel hardware architectures, exemplified by general purpose graphics processing units (GPUs), whose memory bandwidth and computational throughput has rapidly outpaced that of CPUs over the last decade (Figure 4.1). Exploiting this increased power is a tantalizing prospect for many combinatorial problems in statistical NLP, such as parsing (Canny et al., 2013; Hall et al., 2014), speech recognition (Chong et al., 2009, 2008), and phrase extraction for machine translation (He et al., 2015). As these efforts have shown, it is not trivial to exploit this computational power, because the GPU computational model rewards data parallelism, minimal branching, and minimal access to global memory, patterns ignored by many classic NLP algorithms (Section 4.1).

This chapter presents the first language model data structure designed for this computational model. My data structure is a trie in which individual nodes are represented by B-trees, which are searched in parallel (Section 4.2) and arranged compactly in memory in order to fit on the GPU (Section 4.4). My experiments across a range of
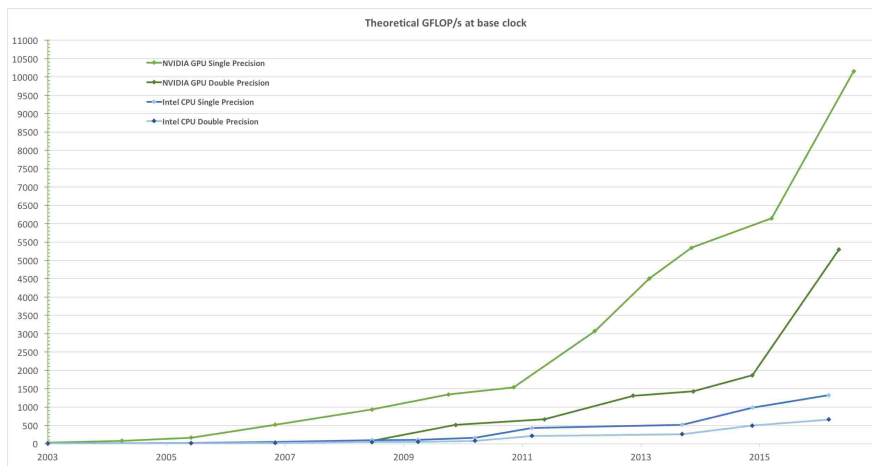
Figure 4.1: Theoretical floating point performance of CPU and GPU hardware over time (NVIDIA Corporation, 2015).

parameters in a batch query setting show that this design achieves a throughput six times higher than KenLM (Heafield, 2011), a highly efficient CPU implementation (Section 4.5). They also show the effects of device saturation and of data structure design decisions.

4.1 GPU computational model

This section builds on Chapter 2 and delivers an in depth description of the GPU architecture (Figure 4.2). It is essential to have an in depth understanding of the GPU architecture in order to design efficient software for it.

GPUs and other parallel hardware devices have a different computational profile from general purpose CPU architectures such as x86, ARM or PowerPC. As discussed in Chapter 2, data structures designed for serial models of computation are not appropriate for massively parallel hardware.

4.1.1 GPU design

Physically, a GPU consists of many simple computational **cores**, which have neither complex caches nor branch predictors to hide latencies. This makes them inherently slower than CPUs, but because they have far fewer circuits than CPU cores, GPU cores are much smaller, and many more of them can fit on a device. So the higher throughput of a GPU is due to the sheer number of cores, each executing a single **thread** of computation (Figure 4.2). Each core belongs to a **Streaming Multiprocessor (SM)**,
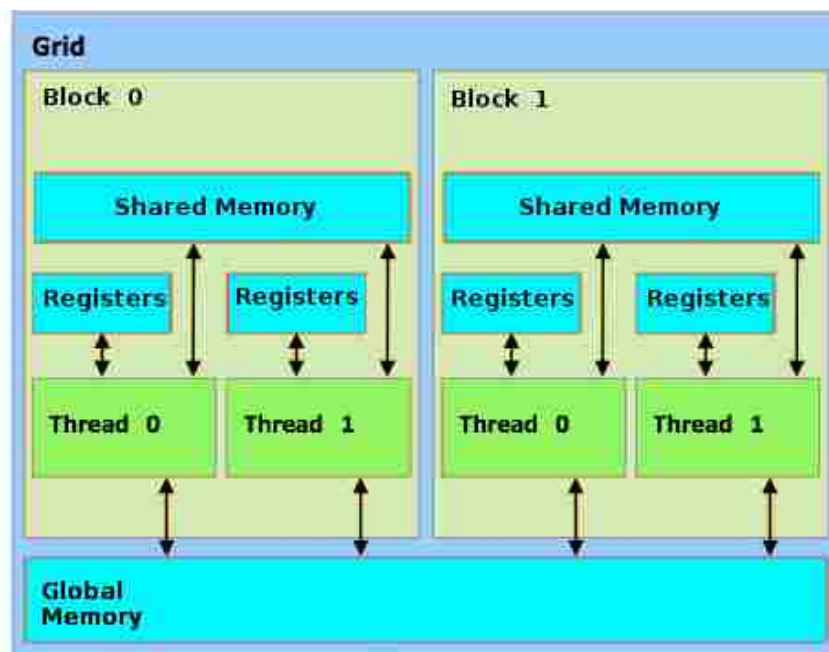
Figure 4.2: GPU memory hierarchy and computational model (NVIDIA Corporation, 2015).

and all cores belonging to a SM must execute the same instruction at each time step, with exceptions for branching described below. This execution model is very similar to single instruction, multiple data (SIMD) parallelism.[1]

Computation on a GPU is performed by an inherently parallel function or **kernel**, which defines a **grid** of data elements to which it will be applied, each processed by a **block** of parallel threads. Once scheduled, the kernel executes in parallel on all cores allocated to blocks in the grid. At minimum, it is allocated to a single **warp**—32 cores on my the GPU I use for experiments. If fewer cores are requested, a full warp is still allocated, and the unused cores idle. Hence in order to achieve to full theoretical performance of the GPU, it's important to use all the cores.

As we have seen with other hardware, the GPU offers several memory types, which differ in size and latency (Table 4.1). Just like a CPU program, being careless with memory accesses patterns will be detrimental performance. Unlike a CPU program, which can treat memory abstractly, a GPU program must explicitly specify in which physical memory each data element resides. This choice has important implications for efficiency that entail design tradeoffs, since, as we have already seen, memory closer to a core is small and fast, while memory further away is large and slow (Table 4.1).

---

[1]Due to differences in register usage and exceptions for branching, this model is not pure SIMD. Nvidia calls it SIMT (single instruction, multiple threads).

Memory type	Latency	Size
Register	0	4B
Shared	4–8	16KB–96KB
Global GPU	200–800	2GB–12GB
CPU	10K+	16GB–1TB

Table 4.1: Latency (in clock cycles) and size of different GPU memory types. Estimates are adapted from NVIDIA Corporation (2015) and depend on several aspects of hardware configuration.

Data structure	Size	Query speed	Ease of backoff	Construction time	Lossless
Trie (Fredkin, 1960)	Small	Medium	No	Fast	Yes
Reverse Trie (Heafield, 2011)	Small	Fast	Yes	Fast	Yes
Probing hash table (Heafield, 2011)	Larger	Faster	Yes	Fast	Yes
Double array (Yasuhara et al., 2013)	Larger	Fastest	Yes	Very slow	Yes
Bloom filter (Talbot and Osborne, 2007)	Small	Slow	No	Fast	No

Table 4.2: A survey of language model data structures and their computational properties.

4.1.2 Designing efficient GPU algorithms

To design an efficient GPU application we must observe the constraints imposed by the hardware, which dictate several important design principles.

Minimize global memory accesses. Data in the CPU memory must first be transferred to the device. This is very slow, so data structures must reside in GPU memory. But even when they reside in global GPU memory, latency is high, so wherever possible, data should be accessed from shared or register memory. This is similar to the situation in the CPU world discussed in Chapter 2: Accessing memory in DRAM is much slower than accessing memory from caches, so I want to minimize it.

Access memory with coalesced reads. In chapter 2, I described the DRAM burst effect, which makes accessing consecutive memory locations much cheaper compared to random memory locations. The same effect also appears in the GPU world: When a thread requests a byte from global memory, it is copied to shared memory along with many surrounding bytes (between 32 and 128 depending on the architecture). So, if consecutive threads request consecutive data elements, the data is copied in a single operation (a coalesced read), and the delay due to latency is incurred only once for all

threads, increasing throughput.

Avoid branching instructions. If a branching instruction occurs, threads that meet the branch condition run while the remainder idle (a **warp divergence**). When the branch completes, threads that don't meet the condition run while the first group idles. So, to maximize performance, code must be designed with little or no branching. This is a problem unique to programming vector processors such as the GPU.

Use small data structures. Another problem unique to the GPU programming is that the available memory is much lower than what we are used to in the CPU world: The maximum amount of memory of a state-of-the-art GPU is 24GB.[2] Language models that run on CPU frequently exceed these sizes, so my data structures must have the smallest possible memory footprint.

4.2 A massively parallel language model

Let w be a sentence, w_i its ith word, and N the order of our model. An N-gram language model defines the probability of w as:

$$P(w) = \prod_{i=1}^{|w|} P(w_i \mid w_{i-1}...w_{i-N+1}) \tag{4.1}$$

A backoff language model (Chen and Goodman, 1999) is defined in terms of n-gram probabilities $P(w_i \mid w_{i-1}...w_{i-n+1})$ for all n from 1 to N, which are in turn defined by n-gram parameters $\hat{P}(w_i...w_{i-n+1})$ and backoff parameters $\beta(w_{i-1}...w_{i-n+1})$. Usually $\hat{P}(w_i...w_{i-n+1})$ and $\beta(w_{i-1}...w_{i-n+1})$ are probabilities conditioned on $w_{i-1}...w_{i-n+1}$, but to simplify the following exposition, I will simply treat them as numeric parameters, each indexed by a reversed n-gram. If parameter $\hat{P}(w_i...w_{i-n+1})$ is nonzero, then:

$$P(w_i \mid w_{i-1}...w_{i-n+1}) = \hat{P}(w_i...w_{i-n+1})$$

Otherwise:

$$P(w_i \mid w_{i-1}...w_{i-n+1}) = P(w_i \mid w_{i-1}...w_{i-n+2}) \times \beta(w_{i-1}...w_{i-n+1})$$

This recursive definition means that the probability $P(w_i \mid w_{i-1}...w_{i-N+1})$ required for Equation 4.1 may depend on multiple parameters. If $r \ (< N)$ is the largest value for

---

[2]In practice GPUs with more memory do exist, but they are prohibitively expensive and available only as limited edition devices.

which $\hat{P}(w_i \mid w_{i-1}...w_{i-r+1})$ is nonzero, then we have:

$$P(w_i \mid w_{i-1}...w_{i-N+1}) = \hat{P}(w_i...w_{i-r+1}) \prod_{n=r+1}^{N} \beta(w_{i-1}...w_{i-n+1}) \qquad (4.2)$$

My data structure must be able to efficiently access these parameters.

4.2.1 Language model data structures

With this computation in mind, I surveyed several popular data structures that have been used to implement *N*-gram language models on CPU, considering their suitability for adaptation to GPU (Table 4.2):

Trie (Fredkin, 1960) is the most widely used data structure for a language model. Instead of string full phrases, it stores common prefixes together to save space. They are used in SRILM and IRSTLM. Trie-based data structures are the smallest non-lossy container for *n*-gram language models.

the house is red   0.333
the house is green   0.333
the house is blue   0.333
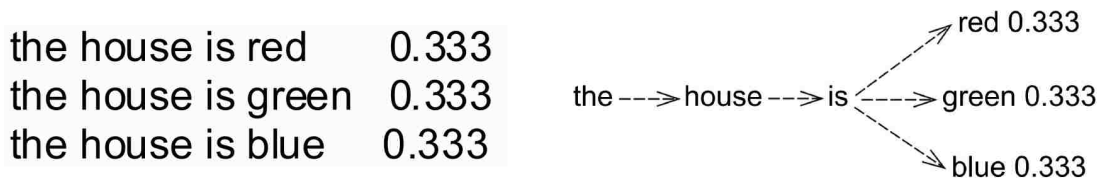
Figure 4.3: Store common substrings together, collapsing "the house is" together.

Reverse trie is an optimization over tries where all phrases are store backwards. This helps during query time because it enables backoff probability to be extracted without additional queries. This data structure is used in KenLM and IRSTLM. The reason why it is faster has to do with the backoff calculation, which is explained in detail in section 4.3.

Probing Hash table is another backend used in KenLM. It is based on a the same probing hash table, as the one I used in Chapter 3: The whole phrase is hashed so only one query to the data structure per *n*-gram is necessary to extract a given score and in case there is a need to backoff, previous *n*-gram queries' results are cached in order to avoid unnecessary searches. This contrasts with the pointer chasing that needs to be done in a reverse trie. This approach is nearly twice as fast as the reverse trie (Heafield, 2011), but consumes more memory, an undesirable quality for the limited memory availability on the GPU.
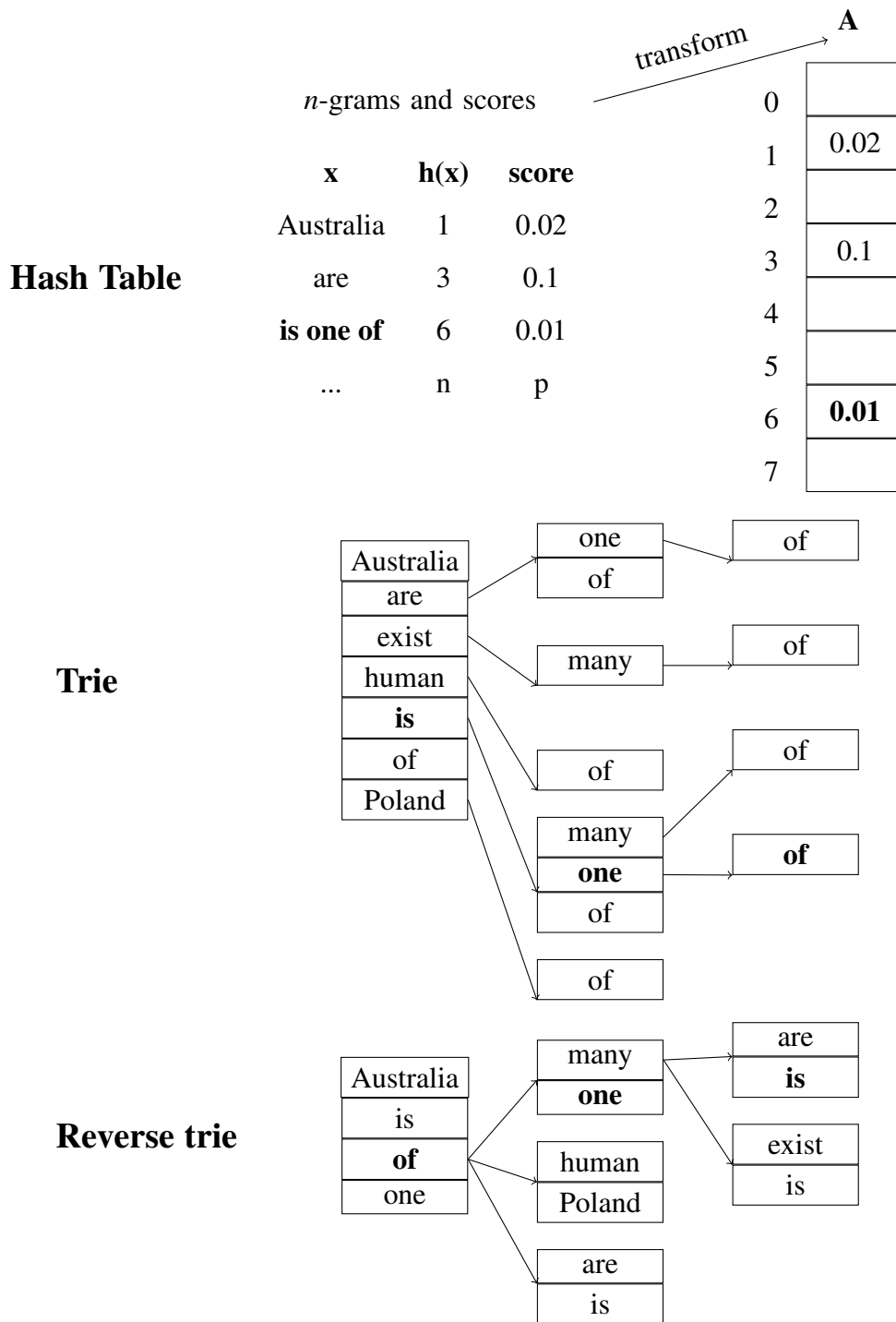
Figure 4.4: A fragment of language model, implemented using a hash table, a trie and a reverse trie, showing the query of N-gram **is one of** in bold. If we query for the N-gram **every one of**, which is not in my example, we would need to fetch the longest match $\hat{P}(\textbf{of} \mid \textbf{one})$ and the backoff parameter $\beta(\textbf{every one})$ (not shown on the table).In the hash table case we would start to query the hash array **A** incrementally from unigrams and rely on cache to find the longest match in case of missing n-gram.In the trie case, we would need to recursively have to go back to the root, querying first, **every one of**, then **one of** and if this is not found, **of**. In addition to that we would need to also fetch β resulting in even more jumps back to the root node. In the reverse trie case we start traversing from the back of the phrase, so in case of a missing n-gram, we would immediately find what is the longest match and we only need to go back to the root once to fetch β. Based on image from Federico et al. (2008).

A contrasting comparison of trie, reverse trie and a hash table is shown on Figure 4.4.

4.2.1.1   Alternative data structures

I introduced the most common data structures that are used in the design of the majority of language model implementations, but I also give a brief survey over another two data structures that deliver the smallest possible model size and the fastest possible query time on the CPU:

Bloom filters RandLM (Talbot and Osborne, 2007) uses randomized bloom filters to store n-grams. Bloom filters are a variation on hash tables which do not resolve collisions. Because of the nature of the data structure used, the model is very compact, but lossy. Furthermore as bloom filters require the input to be hashed with multiple hash functions, more computations need to be performed during retrieval compared to other data structures and thus the performance is lower (Heafield, 2011). On the GPU, the additional computational cost is not of concern as we have vast reserves of computational power, but each hash function denotes an additional random memory access which would make querying slow.

Double arrays DALM (Yasuhara et al., 2013) is a language model based on double arrays (Aoe et al., 1992). Double arrays are an efficient representation of tries, where each n-gram is retrieved after exactly n random memory accesses where n is the order of the n-gram. The data structure is very fast, delivering 5%-10% more queries per second and a smaller model size than KenLM's probing backend. The problem with the double array implementation lies in its construction cost: $O(MN^2)$ where M is the number of trie nodes of a trie representation and N is the number of unique words, which makes it impractical for real world applications. Using the computational power of the GPU to speed up construction is one way to alleviate this issue, but I decided against pursing this line of work, as the data structure is still larger than a trie and conserving memory is a high priority.

Since a small memory footprint is crucial, I implemented a variant of the reverse trie data structure of Heafield (2011). I hypothesized that its slower query speed compared to a probing hash table would be compensated for by the throughput of the GPU, a question I return to in Section 4.5.

4.3 Data structures used in my implementation

In this section I introduce in detail the data structures I have chosen to implement the first GPU n-gram language model.

4.3.1 Reverse trie

A reverse trie language model exploits two important guarantees of backoff estimators: first, if $\hat{P}(w_i...w_{i-n+1})$ is nonzero, then $\hat{P}(w_i...w_{i-m+1})$ is also nonzero, for all $m < n$; second, if $\beta(w_{i-1}...w_{i-n+1})$ is one, then $\beta(w_{i-1}...w_{i-p+1})$ is one, for all $p > n$. Zero-valued n-gram parameters and one-valued backoff parameters are not explicitly stored. To compute $P(w_i|w_{i-1}...w_{i-N+1})$, we iteratively retrieve $\hat{P}(w_i...w_{i-m+1})$ for increasing values of m until we fail to find a match, with the final nonzero value becoming $\hat{P}(w_i...w_{i-r+1})$ in Equation 4.2. We then iteratively retrieve $\beta(w_{i-1}...w_{i-n+1})$ for increasing values of n starting from $r + 1$ and continuing until $n = N$ or we fail to find a match, multiplying all retrieved terms to compute $P(w_i|w_{i-1}...w_{i-N+1})$ (Equation 4.2). The reverse trie is designed to execute these iterative parameter retrievals efficiently.

Let Σ be a our vocabulary, Σ^n the set of all n-grams over the vocabulary, and $\Sigma^{[N]}$ the set $\Sigma^1 \cup ... \cup \Sigma^N$. Given an n-gram **key** $w_i...w_{i-n+1} \in \Sigma^{[N]}$, our goal is to retrieve **value** $\langle \hat{P}(w_i...w_{i-n+1}), \beta(w_i...w_{i-n+1}) \rangle$. We assume a bijection from Σ to integers in the range $1,...,|\Sigma|$, so in practice all keys are sequences of integers.

When $n = 1$, the set of all possible keys is just Σ. For this case, we can store keys with nontrivial values in a sorted array A and their associated values in an array V of equal length so that $V[j]$ is the value associated with key $A[j]$. To retrieve the value associated with key k, we seek j for which $A[j] = k$ and return $V[j]$. Since A is sorted, j can be found efficiently with binary or interpolated search (Figure 4.5).

When $n > 1$, queries are recursive. For $n < N$, for every $w_i...w_{i-n+1}$ for which $\hat{P}(w_i...w_{i-n+1}) > 0$ or $\beta(w_i...w_{i-n+1}) < 1$, our data structure contains associated arrays $K_{w_i...w_{i-n+1}}$ and $V_{w_i...w_{i-n+1}}$. When key k is located in $A_{w_i...w_{i-n+1}}[j]$, the value stored at $V_{w_i...w_{i-n+1}}[j]$ includes the address of arrays $A_{w_i...w_{i-n+1}k}$ and $V_{w_i...w_{i-n+1}k}$. To find the values associated with an n-gram $w_i...w_{i-n+1}$, we first search the root array A for j_1 such that $A[j_1] = w_i$. We retrieve the address of A_{w_i} from $V[j_1]$, and we then search for j_2 such that $A_{w_i}[j_2] = w_{i-1}$. We continue to iterate this process until we find the value associated with the longest suffix of our n-gram stored in the trie. We therefore iteratively retrieve the parameters needed to compute Equation 4.2, returning to the root exactly once if backoff parameters are required. For contrast in the extreme case,

a forward trie will return to the root node at most *n* times where *n* is the context window of the *n*-gram language model.

4.3.2 *K*-ary search

On a GPU, the trie search algorithm described above is not efficient because it makes extensive use of binary search, which is an inherently serial algorithm, as shown in Chapter 2. However, there is a natural extension of binary search that is well-suited to GPU: *K*-ary search (Hwu, 2011), described in detail in Chapter 2. Rather than divide an array in two as in binary search, *K*-ary search divides it into *K* equal parts and performs $K - 1$ comparisons simultaneously (Figure 4.6).

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 | 47 | 53 | 56 | 57 | 60 | 61 | 63 | 66 | 67 | 74 | 78 | 81 |

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 |

| 10 | 12 | 13 | **15** | 17 |

| **15** | 17 |
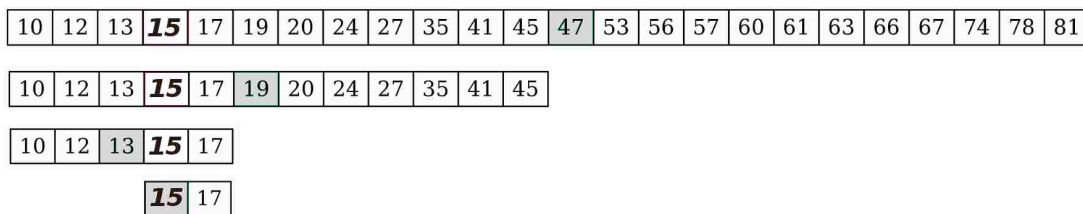
Figure 4.5: Execution of a binary search for key 15. Each row represents a time step and highlights the element compared to the key. Finding key 15 requires four time steps and four comparisons.

| 10 | 12 | 13 | **15** | 17 | 19 | 20 | 24 | 27 | 35 | 41 | 45 | 47 | 53 | 56 | 57 | 60 | 61 | 63 | 66 | 67 | 74 | 78 | 81 |

| **15** | 17 |
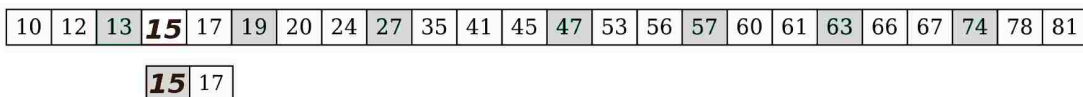
Figure 4.6: Execution of *K*-ary search with the same input as Figure 4.5, for $K = 8$. The first time step executes seven comparisons in parallel, and the query is recovered in two time steps.

To accommodate large language models, the complete trie must reside in global memory, and in this setting, *K*-ary search on an array is inefficient, since the parallel threads will access *K* non-consecutive memory locations, meaning *K* separate memory reads. To avoid this, I require a data structure that places the *K* elements compared by *K*-ary search in consecutive memory locations so that they can be copied from global to shared memory with a single, coalesced read. This data structure is a B-tree (Bayer and McCreight, 1970), which is widely used in databases, filesystems and information retrieval.

4.3.3 B-tree

Informally, a B-tree generalizes binary trees in exactly the same way that *K*-ary search generalizes binary search (Figure 4.7). More formally, a B-tree is a recursive data structure that replaces arrays *A* and *V* at each node of the trie. A B-tree node of size *K* consists of three arrays:

1. A 1-indexed array *B* of $K-1$ keys.

2. A 1-indexed array *V* of $K-1$ associated values so that $V[j]$ is the value associated with key $B[j]$.

3. In case the node is not a leaf, a 0-indexed array *C* of *K* addresses to child B-trees.

The keys in *B* are sorted, and the subtree at address pointed to by child $C[j]$ represents only key-value pairs for keys between $B[j]$ and $B[j+1]$ when $1 \leq j < K$, keys less than $B[1]$ when $j = 0$, or keys greater than $B[K]$ when $j = K$.

To find a key *k* in a B-tree, we start at the root node, and we seek *j* such that $B[j] \leq k < B[j+1]$. If $B[j] = k$ we return $V[j]$, otherwise if the node is not a leaf node we return the result of recursively querying the B-tree node at the address $C[j]$ ($C[0]$ if $k < B[1]$ or $C[K]$ if $k > B[K]$). If the key is not found in array *B* of a leaf, the query fails. Figure 4.7 shows the querying for the key 15:

1. We read in the root of the B-tree into shared memory, taking advantage of the DRAM burst effect to maximize memory throughput and perform *K*-ary search.

2. *K*-ary search is then performed on the root node, using *K* GPU threads in parallel. We identify that the key we are looking for is between 13 and 19 and mark that child node for exploration.

3. We read in the child node in shared memory and perform *K*-ary search again. This time we have identified that the key is located within this node and return the probability and backoff associated with it. In case this trie does not represent a final *n*-gram level, we also extract the address of the root node of the next Trie level.

My complete data structure is a trie in which each node except the root is a B-tree (Figure 4.8). Since the root contains all possible keys, its keys are simply represented by an array *A*, which can be indexed in constant time without any search.

```
struct Entry {
    unsigned int key; //VocabID
    unsigned int next_level; //Offset to next
    float prob;              //trie level
    float backoff;
};
```

Single entry
zoom in

| 13 | 19 | 27 | 47 | 57 | 63 | 74 |

| 10 | 12 | | **15** | 17 | | 20 | 24 | | 35 | 41 | 45 | | 53 | 56 | | 60 | 61 | | 66 | 67 | | 78 | 81 |
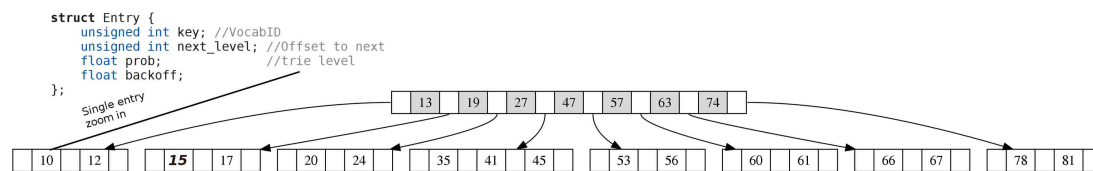
Figure 4.7: In a B-tree, the elements compared in *K*-ary search are consecutive in memory. I also show the layout of an individual entry.
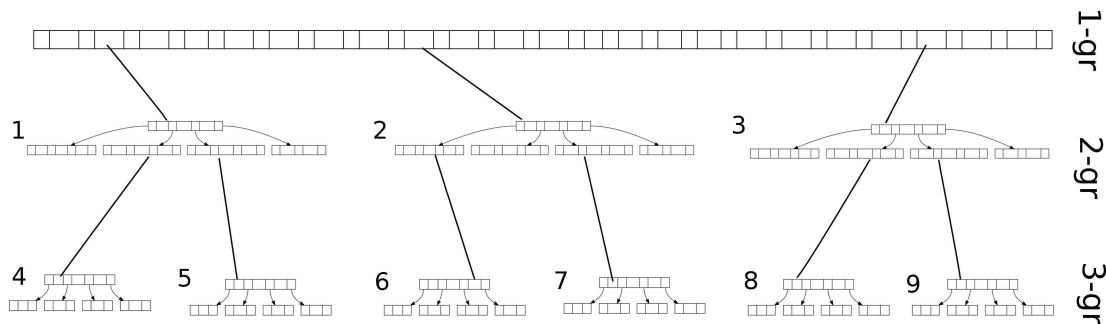
Figure 4.8: Illustration of the complete data structure, showing a root trie node as an array representing unigrams, and nine B-trees, each representing a single trie node. Physically, the entire data structure is laid out in a single consecutive array in memory. The trie nodes are numbered according to the order in which they are laid out.

4.4　Memory layout and implementation

Each trie node represents a unique *n*-gram $w_i...w_{i-n+1}$, and if a B-tree node within the trie node contains key w_{i-n}, then it must also contain the associated values $\hat{P}(w_i...w_{i-n})$, $\beta(w_i...w_{i-n})$, and the address of the trie node representing $w_i...w_{i-n}$ (Figure 4.7). The entire language model is laid out in memory as a single byte array in which trie nodes are visited in breadth-first order and the B-tree representation of each node is also visited in breadth-first order (Figure 4.8). An example *n*-gram query *ABC* on the trie shown on Figure 4.8:

1. We identify the memory location of *A* directly in the unigram array. We can do that because the unigram array contains all vocabulary items and we use the vocabulary ID (an unsigned int) as an offset to encode the memory location. We read in that memory location and there we can find the address of the next trie level that contains all continuations of *n*-grams starting with *A*.

2. We load the root node of the bigram B-tree and perform *K*-ary search in order to find the location of *B*.

3. Once we identify the location of *B* we extract the address for the trigram trie

corresponding to all *n*-grams starting with *AB*.

4. Once again we load up the root node of the trigram trie in memory and perform *K*-ary search in order to find the location of *C* and then extract and return the probability of *ABC*.

Since modern GPUs have a 64-bit architecture, pointers can address 18.1 exabytes of memory, far more than available. To save space, my data structure does not store global addresses; it instead stores the difference in addresses between the parent node and each child. Since the array is aligned to four bytes, these relative addresses are divided by four in the representation, and multiplied by four at runtime to obtain the true offset. This enables encoding relative addresses of 16GB, which is the upper memory limit of the vast majority of devices. I estimate that relative addresses of this size would allow for storing a model containing around one billion *n*-grams.[3] Unlike CPU language model implementations such as those of Heafield (2011) and Watanabe et al. (2009), I do not employ further compression techniques such as variable-byte encoding or LOUDS (Kudo et al., 2011), because their runtime decompression algorithms require branching code, which my implementation must avoid.

I optimize the node representation for coalesced reads by storing the keys of each B-tree consecutively in memory, followed by the corresponding values, also stored consecutively (Figure 4.7). When the data structure is traversed, only key arrays are iteratively copied to shared memory until a value array is needed. This design minimizes the number of reads from global memory.
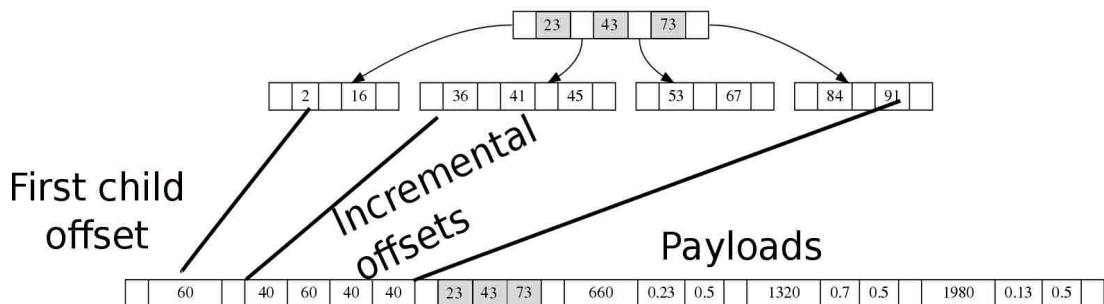
Figure 4.9: Layout of a single B-tree node for $K = 4$. Relative addresses of the four child B-tree nodes (array *C*) are followed by three keys (array *B*), and three values (array *V*), each consisting of an *n*-gram probability, backoff, and address of the child trie node.

---

[3]I estimate this by observing that a model containing 423M *n*-grams takes 3.8Gb of memory, and assuming an approximately linear scaling, though there is some variance depending on the distribution of the *n*-grams.

4.4.1   Constructing minimum depth B-trees

The canonical B-tree construction algorithm (Cormen et al., 2009) produces nodes that are not fully saturated, which is desirable for B-trees that support insertion. However, my B-trees are immutable, and unsaturated nodes of unpredictable size lead to under-utilization of threads, warp divergence, and deeper trees that require more iterations to query. So, I use a construction algorithm inspired by Cesarini and Soda (1983) and Rosenberg and Snyder (1981). It is implemented on CPU, and the resulting array is copied to GPU memory to perform queries.

Since the entire set of keys and values is known in advance for each n-gram, my construction algorithm receives them in sorted order as the array A described in Section 4.2.1. The procedure then splits this array into K consecutive subarrays of equal size, leaving $K - 1$ individual keys between each subarray.[4] These $K - 1$ keys become the keys of the root B-tree. The procedure is then applied recursively to each subarray. When applied to an array whose size is less than K, the algorithm returns a leaf node. When applied to an array whose size is greater than or equal to K but less than $2K$, it splits the array into a node containing the first $K - 1$ keys, and a single leaf node containing the remaining keys, which becomes a child of the first.

4.4.2   Batch queries

To fully saturate the GPU we need to execute many queries simultaneously. A grid receives the complete set of N-gram queries and each block processes a single query by performing a sequence of K-ary searches on B-tree nodes.

4.5   Experiments

I compared my open-source GPU language model **gLM** with the CPU language model KenLM (Heafield, 2011).[56] KenLM can use two quite different language model data structures: a fast probing hash table, and a more compact but slower trie, which inspired my own language model design. Except where noted, the B-tree node size $K = 31$, and I measure throughput in terms of query speed, which does not include the

---

[4]Since the size of the array may not be exactly divisible by K, some subarrays may differ in length by one.

[5]https://github.com/XapaJIaMnu/gLM

[6]https://github.com/kpu/kenlm/commit/9495443

cost of initializing or copying data structures, or the cost of moving data to or from the GPU. Unfortunately, I could not perform any Machine translation experiments, because there is no statistical machine translation decoder on the GPU, so batch perplexity calculation is the only benchmark I could perform.

I performed my GPU experiments on an Nvidia Geforce GTX Titan Black, a state-of-the-art GPU, released in the first quarter of 2015 and costing 1000 USD. The CPU experiments were performed on two different devices: one for single-threaded tests and one for multi-threaded tests. For the single-threaded CPU tests, I used an Intel Quad Core i7 4720HQ CPU released in the first quarter of 2015, costing 280 USD, and achieving 85% of the speed of a state-of-the-art consumer-grade CPU when single-threaded. For the multi-threaded CPU tests I used two Intel Xeon E5-2680 CPUs, offering a combined 16 cores and 32 threads, costing at the time of their release 3,500 USD together. Together, their performance specifications are similar to the recently released Intel Xeon E5-2698 v3 (16 cores, 32 threads, costing 3,500USD). The different CPU configurations are favorable to the CPU implementation in their tested condition: the consumer-grade CPU has higher clock speeds in single-threaded mode than the professional-grade CPU; while the professional-grade CPUs provide many more cores (though at lower clock speeds) when fully saturated. Except where noted, CPU throughput is reported for the single-threaded condition.

Except where noted, my language model is the Moses 3.0 release English 5-gram language model, containing 88 million *n*-grams.[7] The benchmark task computes perplexity on data extracted from the Common Crawl dataset used for the 2013 Workshop on Machine Translation, which contains 74 million words across 3.2 million sentences.[8] Both gLM and KenLM produce identical perplexities, so I am certain that my implementation is correct. Except where noted, the faster KenLM Probing backend is used. The perplexity task has been used as a basic test of other language model implementations (Osborne et al., 2014; Heafield et al., 2015).

4.5.1 Query speed

When compared to single-threaded KenLM, my results (Table 4.3) show that gLM is over six times faster than the fast probing hash table, and nearly fifteen times faster than the trie data structure, which is quite similar to my own, though slightly smaller

---

[7]http://www.statmt.org/moses/RELEASE-3.0/models/fr-en/lm/europarl.lm.1
[8]http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz

LM (threads)	Throughput	Size (GB)
KenLM probing (1)	10.3M	1.8
KenLM probing (16)	49.8M	1.8
KenLM probing (32)	**120.4M**	1.8
KenLM trie (1)	4.5M	**0.8**
gLM	65.5M	1.2

Table 4.3: Comparison of gLM and KenLM on throughput (*N*-gram queries per second) and data structure size.

due to the use of compression.

When I fully saturate the professional-grade CPU, using all sixteen cores and sixteen hyperthreads, KenLM is about twice as fast as gLM. However, the CPU costs nearly four times as much as the GPU, so economically, this comparison favors the GPU.

On first glance, the KenLM's scaling from one to sixteen threads is surprisingly sublinear. This is not due to vastly different computational power of the individual cores, which are actually identical. It is instead due to scheduling, cache contention, and—most importantly—the fact that the CPUs implement *dynamic overclocking*: the base clock rate of 2.7 GHz at full saturation increases to 3.5 GHz when the professional CPU is underutilized, as when single-threaded; the rates for the consumer-grade CPU similarly increase from 2.6 to 3.6 GHz.[9]

4.5.2   Effect of B-tree node size

What is the optimal K for the B-tree node size? I hypothesized that the optimal size would be one that approaches the size of a coalesced memory read, which should allows to maximize parallelism while minimizing global memory accesses and B-tree depth. Since the size of a coalesced read is 128 bytes and keys are four bytes, I hypothesized that the optimal node size would be around $K = 32$, which is also the size of a warp. I tested this by running experiments that varied K from 5 to 59, and the results (Figure 4.10) confirmed my hypothesis. As the node size increases, throughput increases until a node size of 33 is reached, where it steeply drops. This result highlights the importance of designing data structures that minimize global memory access and maximize parallelism.

---

[9]Intel calls this Intel Turbo Boost.
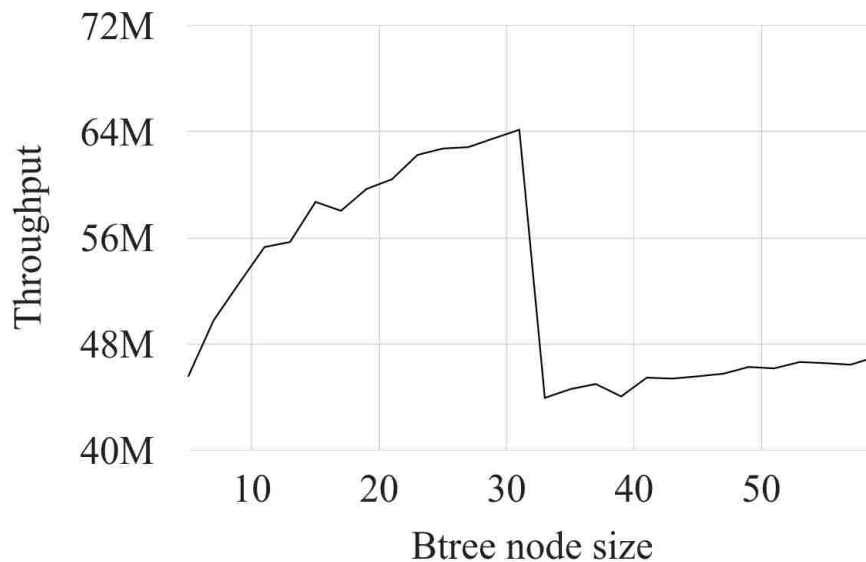
Figure 4.10: Effect of BTree node size on throughput (*n*-gram queries per second)

I was curious about what effect this node size had on the depth of the B-trees representing each trie node. Measuring this, I discovered that for bigrams, 88% of the trie nodes have a depth of one—I call these **B-stumps**, and they can be exhaustively searched in a single parallel operation. For trigrams, 97% of trie nodes are B-stumps, and for higher order *n*-grams the percentage exceeds 99%. To put this in perspective: it means that in the vast majority of cases, the GPU can find the relevant data (if it exists) in a single fetch, followed by a simultaneous comparison of every value in the array.

4.5.3 Saturating the GPU

A limitation of my approach is that it is only effective in high-throughput situations that continually saturate the GPU. In situations where a language model is queried only intermittently or only in short bursts, a GPU implementation may not be useful. I wanted to understand the point at which this saturation occurs, so I ran experiments varying the batch size sent to my language model, comparing its behavior with that of KenLM. To understand situations in which the GPU hosts the language model for query by an external GPU, I measure query speed with and without the cost of copying queries to the device.

My results (Figure 4.11) suggest that the device is nearly saturated once the batch size reaches a thousand queries, and fully saturated by ten thousand queries. Throughput remains steady as batch size increases beyond this point. Even with the cost of copying batch queries to GPU memory, throughput is more than three times higher than
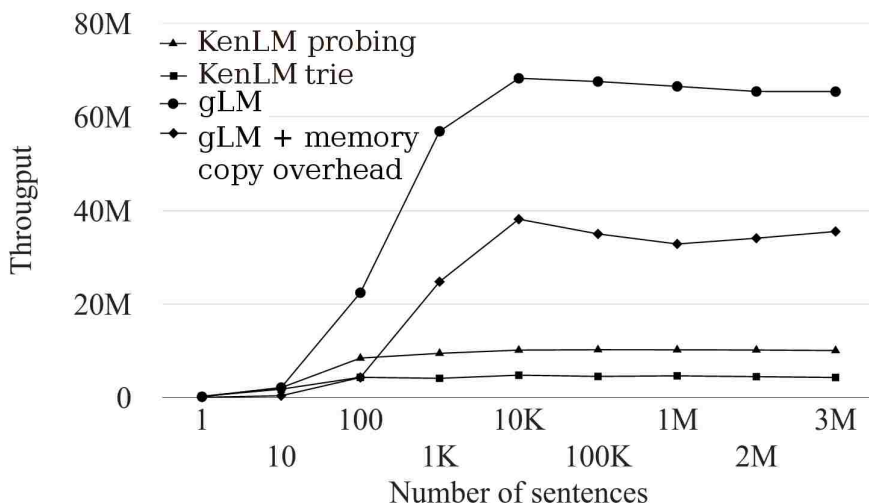
Figure 4.11: Throughput (*N*-gram queries per second) vs. batch size for gLM, KenLM probing, and KenLM trie.

	Regular LM	Big LM
KenLM	10.2M	8.2M
KenLM Trie	4.5M	3.0M
gLM	**65.5M**	**55M**

Table 4.4: Throughput comparison (*n*-gram queries per second) between **gLM** and **KenLM** with a 5 times larger model and a regular language model.

that of single threaded KenLM. I have not included results of multi-threaded KenLM scaling on Figure 4.11 but they are similar to the single-threaded case: throughput (as shown on Table 4.3) plateaus at around one hundred sentences per thread.

4.5.4   Effect of model size

To understand the effect of model size on query speed, I built a language model with 423 million *n*-grams, five times larger than my basic model. The results (Table 4.4) show an 18% slowdown for gLM and 20% slowdown for KenLM, showing that model size affects both implementations similarly.

4.5.5   Effect of *N*-gram order on performance

All experiments so far use an *N*-gram order of five. I hypothesized that lowering the *n*-gram order of the model would lead to faster query time (Table 4.5). I observe that *N*-gram order affects throughput of the GPU language model much more than the

	5-gram	4-gram	3-gram
KenLM	10.2M	9.8M	11.5M
KenLM Trie	4.5M	4.5M	5.2M
gLM	**65.5M**	**71.9M**	**93.7M**

Table 4.5: Throughput comparison (*n*-gram queries per second) achieved using lower order *n*-gram models.

CPU one. This is likely due to effects of backoff queries, which are more optimized in KenLM. At higher orders, more backoff queries occur, which reduces throughput for gLM.

Effect of templated code   My implementation initially relied on hard-coded values for parameters such as B-tree node size and *N*-gram order, which I later replaced with parameters. Surprisingly, I observed that this led to a reduction in throughput from 65.6 million queries per second to 59.0 million, which I traced back to the use of dynamically allocated shared memory, as well as compiler optimizations that only apply to compile-time constants. To remove this effect, I heavily templated my code, using as many compile-time constants as possible, which improves throughput but also allows for changes in parameters without recompilation.

4.5.6   Bottlenecks: computation or memory?

On CPU, language models are typically memory-bound: most cycles are spent in random memory accesses, with little computation between accesses. To see if this is true in gLM I experimented with two variants of the benchmark in Figure 4.3: one in which the GPU core was underclocked, and one in which the memory was underclocked. This effectively simulates two variations in my hardware: A GPU with slower cores but identical memory, and one with slower memory, but identical processing speed. I found that throughput decreases by about 10% when underclocking the cores by 10%. On the other hand, underclocking memory by 25% reduced throughput by 1%. I therefore conclude that gLM is computation-bound, which is good for future proofing, because communication improves much slower than computation with newer hardware generations. I expect therefore that gLM will continue to improve on future generations of parallel devices offering higher theoretical floating point performance without the need of code modifications.

4.6   Conclusion and Future work

I have designed the first *n*-gram language model for a vector processor. I have shown that in spite of the architecture being different, the underlying rule is still the same: in order to extract maximum performance we need to minimize memory accesses. My implementation and code are specific to the GPU, but the design, data structures and much of the actual code are likely to be useful to other hardware that supports SIMD parallelism, such as the Xeon Phi.

I designed my GPU language model with the goal to use it eventually in a statistical machine translation decoder. At the time of designing it there was already work on GPU phrase table (He et al., 2013, 2015), so the only component remaining to build a bare bones SMT decoder would be the search (Table 4.6). I was aware of related GPU work on dynamic programming problems (Canny et al., 2013; Hall et al., 2014) and as well as GPU based speech recognition systems (Chong et al., 2009, 2008), which I intended to use as a starting point for implementing search.

	CPU	GPU
Language Model	✓	✓
Translation Model	✓	✓
Decoding	✓	✗

Table 4.6: Novelty matrix for MT components on the GPU

Although phrase-based search on a GPU is an interesting problem with some ongoing work (Argueta and Chiang, 2017, 2018), much of the field shifted in the direction of neural machine translation and because of that I abandoned my ambitions to design a phrase-based decoder on the GPU. Nevertheless a standalone GPU based *n*-gram language model could still be useful.

Neubig and Dyer (2016) show how to interpolate a *n*-gram language model and an RNN language model, as the former models rare words much better than the neural network solution. I could use my implementation to efficiently integrate an *n*-gram language model in a neural one and use that to explore LM interpolation within neural machine translation systems similar to the work of Gülçehre et al. (2015).

Chapter 5

Accelerating Asynchronous Stochastic Gradient Descent on distributed hardware

As discussed in Chapter 4, statistical machine translation became largely superseded by neural machine translation. Neural machine translation training is very computationally expensive, often taking weeks on multiple GPUs. In this chapter, I present methods to accelerate neural network training on multiple GPUs and use machine translation as a case study.

Gradient descent is the most popular way to train neural networks. Gradient descent aims to minimise the error when the neural network predicts labels. Since neural networks are very good at approximating functions, if we run SGD for long enough we will overfit the model to the training sit and make it useless for predicting unseen data. To prevent overfitting, a strategy commonly known as early stopping is used, where the cross-entropy on the validation set is periodically examined and if it stops decreasing, the training is halted. There are three main variants of gradient descent:

Batch gradient descent where we first compute the gradient of a cost function C (typically cross-entropy) with regards to the parameter θ for the entire training dataset D and then update θ based on the gradient and some learning rate η:

$$\theta = \theta - \eta \times \nabla_\theta C(\theta(D)) \tag{5.1}$$

The crucial thing to note here is that we need to compute the gradient for the whole dataset (also known as one full **epoch**), before doing a single gradient update, which brings several drawbacks (Masters and Luschi, 2018):

- It consumes vast amounts of memory for large datasets, as the whole dataset needs to be held in memory in order to compute its gradient. This could be overcome by a technique of delaying gradient updates, described in 5.2.3, where the dataset is split into chunks and its gradient is incrementally computed.

- It could lead to the model getting stuck in saddle points, due to the non-convex nature of neural network optimization. In a non-convex problems we want an optimizer that introduces extra noise during training so that saddle points can be overcome.

- It also doesn't allow on-the-fly model updates, such as updating the model when more data becomes available or fine tuning it.

Methods such as delayed gradient updates, which I introduce later in this chapter, could help with the memory issue of batch gradient descent, but the other shortcomings still stand.

Online stochastic gradient descent addresses the shortcomings of batch gradient descent by updating θ after each training example x:

$$\theta = \theta - \eta \times \nabla_\theta C(\theta(x)) \tag{5.2}$$

This means that there ample training flexibility and there is no longer an issue with memory usage or on-the-fly model updates. Furthermore online gradient descent offers offers faster convergence, due to updating the parameters more often and putting the model in a better place before evaluating the next example. This is also helpful with the non-convex optimization, because there is extra noise introduced during training that could overcome saddle points. However there is another set of shortcomings with this optimization method: It is inefficient in terms of computation due to the frequent updates to the parameters; if the training data has high variance, it has the potential to push the parameters in a bad place.

Mini-batch stochastic gradient descent attempts to compromise and bring the best out of both extremes described above. It groups the training data into **mini-batches** of n datapoints, computes and sums the gradient over them and then updates the model parameters:

$$\theta = \theta - \eta \times \nabla_\theta C(\theta(x_i..x_n)) \tag{5.3}$$

Mini-batch SGD has several key advantages:

- It maps very well to vector processors as mini-batches form a conveniently large matrix which maps efficiently to vector processors, but not too large as to not fit on the device memory.

- Training on data with high variance is a lot more robust than online SGD, as the many datapoints in a mini-batch have a smoothing effect.

- Mini-batches help escape saddle points during training (Ge et al., 2015).

SGD is by far the most widely used variant of gradient descent for neural machine translation training because it works very well in practise and maps efficiently to the hardware.

With training times measured in days, parallelizing SGD is valuable for making experimental progress and scaling data sizes. Modern systems contain up to ten GPUs and there is not enough parallel computation available in recurrent neural networks to allow for model parallelism to saturate the available hardware. We need to do data parallelism, training mini-batches in parallel on differet workers and there are two options to do so:

Synchronous SGD where we have multiple workers processing mini-batches in parallel and when all workers finish their computations, all gradients are summed up together and the model parameters are updated. In effect synchronous SGD simulates training with n times larger mini-batch size where n is the number of workers used. The disadvantage of using synchronous SGD is that often workers need to idle waiting for everyone to complete their computations before being able to proceed with the next mini-batch, leading to sub-optimal performance. This is especially problematic in settings where mini-batches can have varying sizes and neural machine translation is such a setting. The reason is that in the neural machine translation case, mini-batches are formed of parallel sentences of natural language, and sentences in natural language come in varying sizes. Even if we try to form mini-batches of similar length by targeting number of words instead of number of sentences in the mini-batch construction strategy, it is not possible to achieve completely identical mini-batches. This issue is looked at in depth in Section 5.5.1.

Asynchronous SGD where we have multiple devices processing mini-batches in parallel and when each worker finishes its computation, it updates the model, avoiding explicit synchronisation and thus worker idling. As shown on Table 5.1, this

setting allows for nearly linear increase in training speed with the increase of number of GPUs, unlike synchronous SGD. Unfortunately asynchronous SGD brings the **stale gradient** problem (Figure 5.1), where a worker can update the neural network parameters based on a gradient computed against an older version of those parameters (Abadi et al., 2016). This is effect is harmful to the model convergence and potentially requires more epochs, before convergence, negating any speed benefits over synchronous SGD.
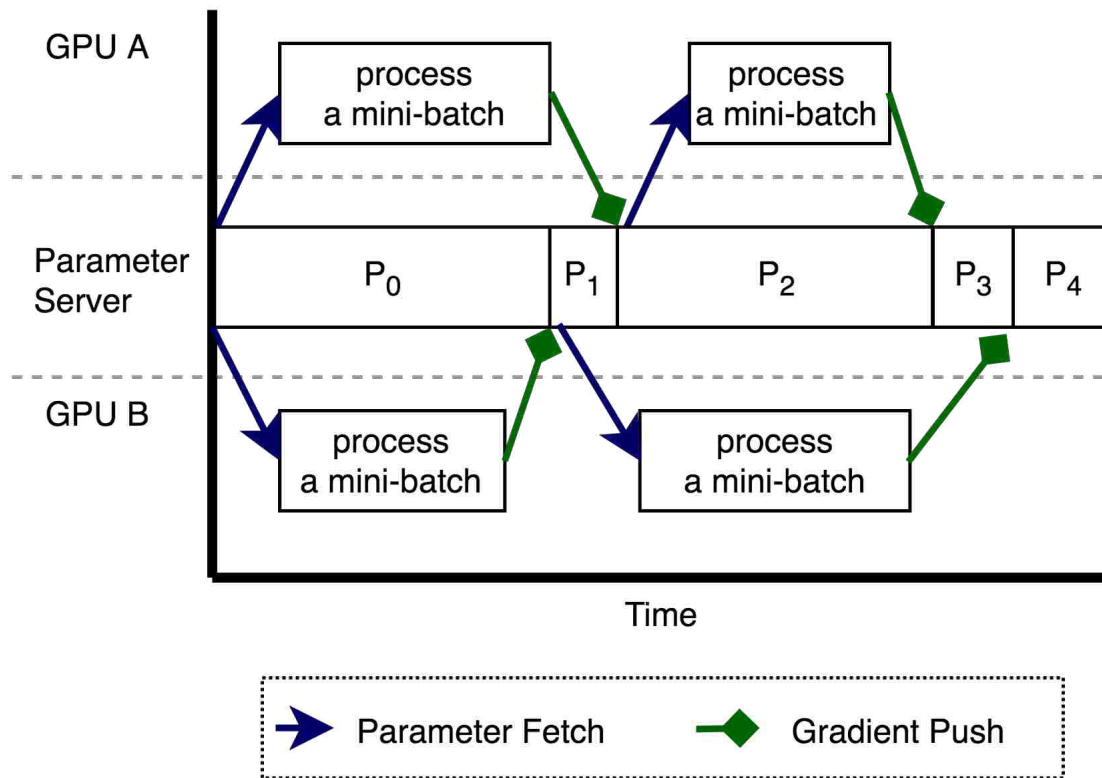
Figure 5.1: Gradient staleness in asynchronous SGD. By the time GPU A submits its gradient, the underlying parameters *P* may have changed, because GPU B has already updated the model with its gradient. This means GPU A's gradient is stale. In a similar manner, B's gradient is stale for the second mini-batch.

Goyal et al. (2017) train ImageNet (Deng et al., 2009), in one hour by using 256 GPUs in parallel and increasing the mini-batch size to reduce communication. In this chapter I explore the applicability of the work of Goyal et al. (2017) in the neural machine translation setting. Neural machine translating differs from the ImagetNet task in several key aspects:

- Goyal et al. (2017) uses a variant of ResNet-50 (He et al., 2016), which is a convolutional neural networks with 25.6M parameters (Zagoruyko and Komodakis,

2016). NMT uses recurrent neural networks with around 200M parameters, nearly an order of magnitude more.[1] As such NMT systems consume a lot more GPU memory and more computational resources for processing a mini-batch.

- Mini-batch sizes in neural machine translation consume a lot more memory due to parallel sentences and their embeddings consuming more memory than images, because the sequences and their embeddings are larger than the image convolutions. This makes it impossible to fit mini-batches of similar magnitude as the ones in Goyal et al. (2017) as we quickly run out of memory (Table 5.1). In addition to that, due to variable-length sentences, machine translation systems commonly fix a memory budget then pack as many sentences as possible into a dynamically-sized batch (Junczys-Dowmunt et al., 2018). As such, the mini-batch size is controlled by the memory amount rather than the number of training examples.

After considering the above and the training speed at different number of GPUs (Table 5.1), I chose asynchronous SGD for my experiments. This method provides nearly double the training speed of synchronous SGD, even if it has convergence issues due to the stale gradient problem. I aim to overcome the convergence problems using a variety of methods and benefit entirely from the higher training speed.

GPUs	Variant	VRAM	Words per Batch	WPS
1	n/a	3 GB	3080	8.3k
1	n/a	7 GB	7310	9.5k
4	Synchronous	3 GB	3080	15.7k
4	Asynchronous	3 GB	3080	19.5k
4	Synchronous	7 GB	7310	23.5k
4	Asynchronous	7 GB	7310	36.6k
4	Asynchronous	10 GB	10448	40.2k

Table 5.1: Relationship between the GPU Memory (VRAM) budget for batches, number of source words processed in each batch and words-per-second (WPS) measured on a shallow model for different number of GPUs and different variants of SGD.

---

[1]Measurement taken from the models of Haddow et al. (2018).

5.1   Parallel asynchronous SGD

I proceed to give a detailed overview of the asynchronous SGD implementation I used: Similar to Dean et al. (2012), it uses data parallelism in which each GPU computes a gradient on part of the data. Each GPU contains one full copy of the model parameters, which it uses to process a mini-batch and in addition to that it contains a fraction of the **parameter server**, which contains the up-to-date model parameters (the local copy of the model might be stale, due to the stale gradient problem described earlier). For efficiency, the parameter server is **sharded** over all GPUs, meaning it is split in equal parts across them. Thus each GPU acts both as a worker, processing mini-batches; and as a server, distributing serving $1/N$th of the parameters, where N is the number of GPUs. The architecture is described on Figure 5.2. Mini-batches processing goes as follows:

- A worker receives a new mini-batch for processing and fetches an up-to-date model from the parameter server. This involves transferring $\frac{N-1}{N}m$ bytes from remote GPUs where m is the model size in bytes. This many-to-one operation corresponds to 'Parameter Fetch' in Figure 5.1.

- The worker then uses the local model copy to process the mini-batch and compute a gradient.

- The worker uses an optimizer (in my case Adam: Kingma and Ba, 2015) to update the parameter server, which again involves $\frac{N-1}{N}m$ bytes of remote GPU communication. This one-to-many operation corresponds to 'Gradient Push' in Figure 5.1.

It is important to note that mutual exclusivity is enforced through locking, so that a shard from the parameter server is never accessed while it's being updated by another thread. This contrasts with the Hogwild method (Recht et al., 2011), where there is no enforced mutual exclusivity for the benefits of speed, but there is no guarantee that all model updates go through.

The more GPUs are involved in the parallel SGD, the more remote GPU communication occurs. Remote GPU communication, as shown in Chapter 2, is many times more expensive than local GPU communication so ideally we want to miminize it. Goyal et al. (2017) made the same observation and this is why they increased the mini-batch size, as that improves the computation to communication ratio.
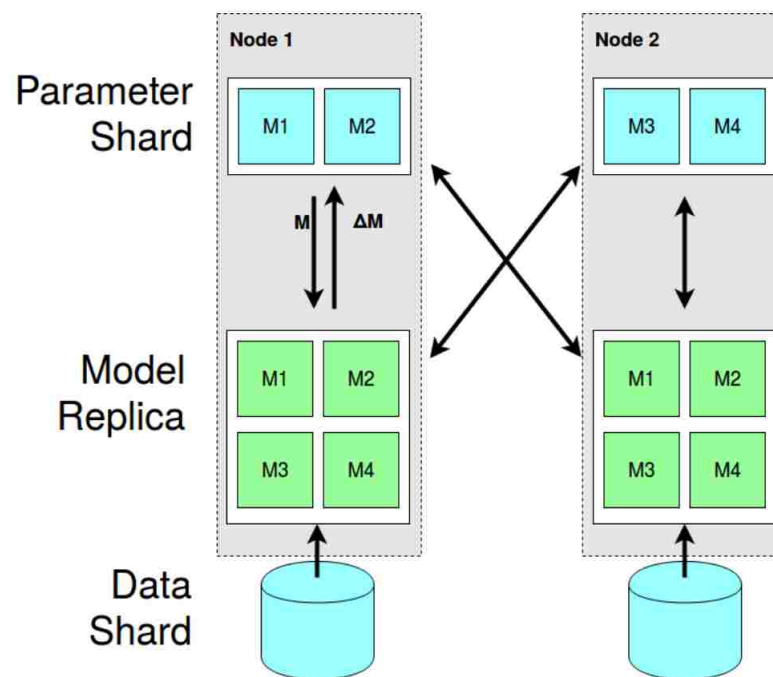
Figure 5.2: Parallel SGD with sharded parameters. Image source: Aji and Heafield (2017)

5.2 Increasing performance with larger batches and delayed updates

In this section, I introduce the methods that I developed to accelerate asynchronous SGD and the model and dataset I used for developing them. My goal is to increase the training speed without sacrificing model quality, or the number of epochs necessary for convergence.

5.2.1 Evaluation metrics

Training speed I measure the speed with which the training set is processed with the **Words-per-Second** (WPS) metric.

Cross Entropy I measure the cross-entropy on the dev set (on all graphs denoted as **CE**) and on the training set (denoted as **CE T**). The cross-entropy on the training set shows whether the model is more general (less overfitted towards the training data) or not. The cross-entropy on the dev set shows how certain the model is when translating the dev set. Since the training set is in general quite large, I used a sample of 10000 sentences from it. The same holds for every other place in this chapter where the

training set cross-entropy is measured.

BLEU I measure the translation quality of the dev set in terms of BLEU (Papineni et al., 2002).

Training time I measure the End-to-End training time for training each system. The end of training is determined by 5 consecutive stalls in the cross-entropy metric of the validation set. Validation frequency is adjusted depending on the mini-batch size so that it happens more often on systems that have larger mini-batches (or delayed gradient updates). This ensures that validation steps happen after roughly the same amount of training data has been processed.

5.2.2   Baseline systems

Dataset I used the Romanian→English task from WMT 16 (Bojar et al., 2016) with backtranslated data, resulting in 2.6 million parallel sentences.

Hardware and Software I used 4 Tesla P 100 GPUs in a single node with the Marian NMT framework for training (Junczys-Dowmunt et al., 2018).

I trained five separate baseline systems: three systems that use vanilla synchronous or asynchronous SGD and two systems based on the work of Aji and Heafield (2017) which also aims to increase training speed:

Asynchronous baseline This system is equivalent to that of Sennrich et al. (2016), which was the first place constrained system (and tied for first overall in the WMT16 shared task). By equivalent I mean that the model and hyperparameters are the same: shallow bidirectional GRU (Cho et al., 2014) encoder-decoder , but it is trained using the Marian machine translation toolkit (Junczys-Dowmunt et al., 2018), as opposed to Nematus (Sennrich et al., 2017b). The memory allowance for mini-batches in my system is 3 GB (for an average batch size of 2633 words). Adam (Kingma and Ba, 2015) is used to perform asynchronous SGD with learning rate of 0.0001.

Synchronous Baseline I also compare with a second baseline system that uses synchronous SGD for training. It differs slightly in hyperparameters, using non-default Adam parameters ($\beta_1 = 0.9, \beta_2 = 0.98$), warmup of 16000 mini-batches and inverse

square root cooldown following which are the recommended settings from Vaswani et al. (2017) for training a NMT system using synchronous SGD.

Optimized Asynchronous baseline   Since I apply optimizations over asynchronous SGD, I performed a learning rate and mini-batch-size parameter sweep over the asynchronous baseline system and settled on a learning rate of 0.00045 and 10 GB memory allowance for mini-batches (average batch size of 10449 words). This is the fastest system I could train without sacrificing translation performance and without changing the codebase. In my experiments, I refer to this system as "Optimized asynchronous".

Asynchronous + gradient dropping   This system uses the work of Aji and Heafield (2017) to drop the lower 99% of the gradients based on absolute values in order to drastically reduce inter-GPU communication. I compare this to my work as it is an alternative method to increase training speed by reducing communication.

Optimized asynchronous + gradient dropping   This system is the same as the one described above, except that I use the hyperparameters of the parameter sweeped baseline.

Table 5.2 show each of those systems and their training performance. The system that trains the fastest is the optimized asynchronous where gradient dropping is used, followed by the optimized asynchronous. In the rest of my experiments my goal is to beat these training times without experiencing degradation in BLEU. It is important to note that while the system using gradient dropping does train faster, the speed up effect is smaller when using larger mini-batches (as is the case with the optimized asynchronous system). Furthermore this model has slightly lower BLEU, on top of taking more epochs to converge, which hints that gradient dropping hurts convergence. We observer that the cross-entropy on the training set doesn't always correlated to the cross-entropy on the dev set.

5.2.3   Large mini-batches and delayed gradient updates

I aim to increase speed, in terms words-per-second (WPS), by increasing the batch size. As discussed, larger batches have two well-known impacts on speed: making more use of GPU parallelism and communicating less often.

After raising the batch size to the maximum that fits on the GPU, while leaving space for the model parameters, **I emulate even larger batches by processing mul-**

System	Hours	Epochs	BLEU	CE	CE T	WPS
synchronous	14.3	11	35.3	50.63	20.43	15.7k
asynchronous (0)	12.2	13	35.61	50.47	18.12	19.5k
(0) + Aji and Heafield (2017)	6.23	12	35.16	50.86	18.16	26.9k
optimized asynchronous (1)	**4.97**	**10**	**35.56**	**50.90**	**18.72**	**40.2k**
(1) + Aji and Heafield (2017)	**4.32**	**11**	**35.16**	**52.02**	**19.66**	**41.5k**

Table 5.2: Romanian-English baseline systems. The systems that are bolded will be used for comparison in my next experiments.

tiple mini-batches and summing their gradients locally without sending them to the optimizer (Figure 5.3). I will refer to this strategy as **delayed gradient updates**. The delayed gradient updates strategy requires extra memory equal to the size of the model, as we need to store a gradient for every single parameter, but in return it allows for arbitrary large mini-batch size. It increases speed because communication is reduced (Table 5.3), just as if we were processing a much larger mini-batch, but at a constant memory cost, as opposed to linearly increased memory cost, associated with increasing the mini-batch size.

GPUs	Variant	VRAM	τ	Words per Batch	WPS
4	Asynchronous	3 GB	1	3080	19.5k
4	Asynchronous	7 GB	1	7310	36.6k
4	Asynchronous	10 GB	1	10448	40.2k
4	Asynchronous	20* GB	2	20897	44.2k
4	Asynchronous	30* GB	3	31345	46.0k
4	Asynchronous	40* GB	4	41794	47.6k

Table 5.3: Updated version of Table 5.1, which includes delayed gradient updates configurations (prefixed by *). Synchronous SGD experiments were not performed due to the slower overall training speed.

I introduce parameter τ, which is the number of iterations a GPU performs locally before communicating externally as if it had run one large batch. The Words-per-second (WPS) column on Table 5.3 shows the effect on corpora processing speed when applying delayed gradient updates for different values of τ. When increasing the mini-batch size τ times without touching the learning rate, there are effectively τ times fewer updates per epoch. On the surface, it might seem that these less frequent updates are counterbalanced by the fact that each update is accumulated over a larger batch, and
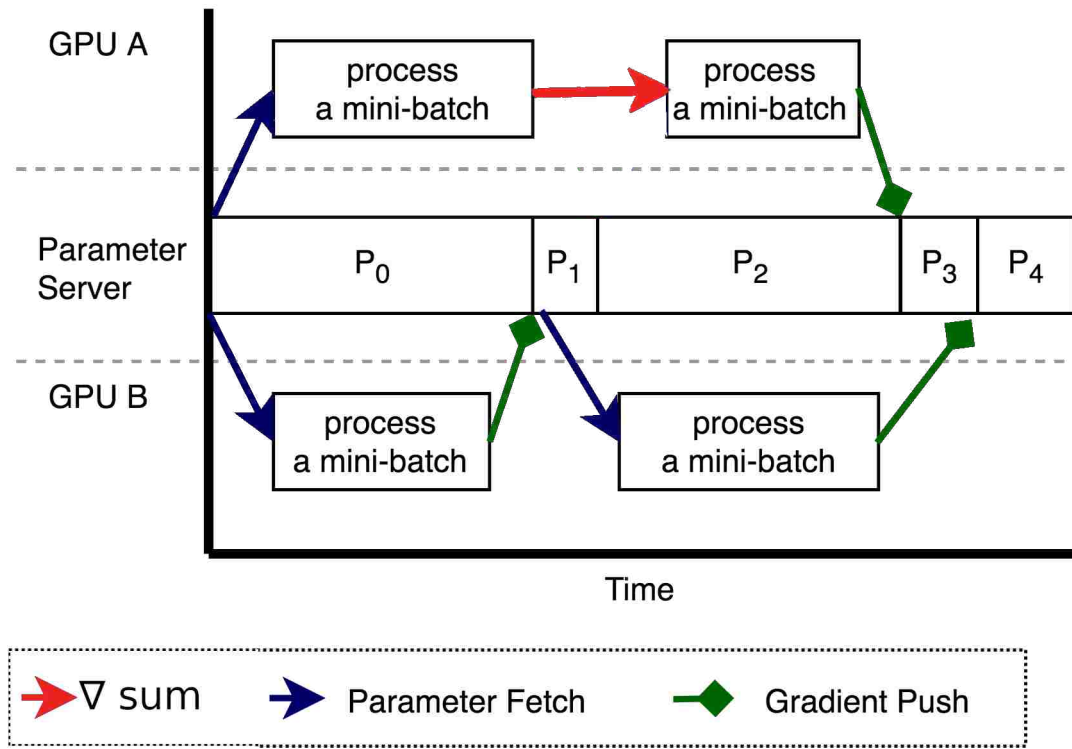
Figure 5.3: Delayed gradient updates, in asynchronous SGD. GPU B performs vanilla SGD, where the model is updated at each timestep, whereas GPU A saves the gradient locally and updates the model only after processing 2 mini-batches.

thus the gradient is bigger. But practical optimization heuristics like gradient clipping limit the maximum absolute value of the gradient, meaning that in effect the model is updated less often, resulting in slower convergence. Goyal et al. (2017) recommend scaling the learning rate linearly with the mini-batch size in order to maintain convergence speed and this is what I did as well: For all the systems, the learning rate is multiplied by the value of τ.

System	Hours	Epochs	BLEU	CE	CE T	WPS
optimized asynchronous (1)	4.97	10	35.56	50.90	18.72	40.2k
(1) + Aji and Heafield (2017)	4.32	11	35.16	52.02	19.66	41.5k
(1) + delayed updates $\tau = 2$ (2)	**4.20**	**11**	**34.82**	**51.68**	**21.53**	**44.2k**

Table 5.4: Romanian-English baseline systems. The systems that are bolded will be used for comparison in my next experiments.

Table 5.4 shows the results of applying $\tau = 2$ delayed gradient updates, as well as learning rate scaling with the same factor to the optimized asynchronous baseline.

Despite sending full gradients, as opposed to only 1%, my model trains faster in terms of words per second. This means that delaying gradient updates is more effective at increasing training speed than the work of Aji and Heafield (2017). While my model does achieve faster training time, it comes at the cost of lower BLEU score and more epochs necessary to converge. Therefore, the increase of training speed when delaying gradient updates comes at the cost of model quality, which is not the desired goal of my optimizations. We observe that delaying gradient updates has better dev set cross-entropy scores than the work of Aji and Heafield (2017) but this is not reflected in better BLEU. Still convergence is worse compared to all other systems, especially when we consider the training set cross-entropy.

5.3 Improving model convergence

I have shown that training speed can be increased by artificially increasing the mini-batch size, however the model then takes more epochs to converge now, which means parts of the benefits of the faster training are lost, which is contrary to my goal. In order to ensure that I have the same convergence rate with my improved training speed, I applied the following techniques:

5.3.1 Warmup

Goyal et al. (2017) point out that just increasing the learning rate performs poorly for very large batch sizes. The reason is that when the model is initialized at a random point, the error after the first updates will be quite large and so would be the gradient. When the learning rate is large, the first updates will push the parameters too much, resulting in lots of jerky updates until the model finds a good point, similar to the illustration on Figure 5.4. However in practise the model frequently can't recover from bad initialization and this leads to suboptimal convergence point. Goyal et al. (2017) suggest that initially model updates should be small so that the model will not be pushed in a suboptimal state.

Lowering initial learning rate   Goyal et al. (2017) lower the initial learning rate and gradually increase it over a number of mini-batches until it reaches a predefined maximum. This technique is also adopted in the work of Vaswani et al. (2017). This is the canonical way to perform warmup for neural network training.

Local optimizers Learning rate warmup addresses the problem of big potentially harmful updates to the model by reducing the learning rate, so that the initial updates to the model carry less weight. I propose an alternative strategy that addresses the core issue of the model not being updated often enough due to the mini-batch size being too large. Since I emulate large batches by running multiple smaller batches, **I can use the intermittently computed gradient to update the model.** I introduce additional worker-local optimizers, that update the local copy of the model between each processed batch. I also update the shard of the global model that happens to be on the same GPU. I maintain the model update procedure described in section 5.1 where a global optimizer run is performed using the summed up gradient.

Local optimizers update the worker-local model over which gradients are produced, which in turn means that every mini-batch that gets processed will see a slightly more up-to-date model and produce a more accurate gradient, which in turn will produce better updates for the global model. Updating the worker-local shard of the global model reduces model staleness, because it allows all workers to perform updates over parameter shards that are updated every mini-batch as opposed to every τ mini-batches. Local updates are very fast, because they don't involve remote device communication, therefore the cost of using local optimizers in my model is negligible.

I use local optimizers purely as a warmup strategy, turning them off after the initial phase of the training. Empirically, I found that I can get the best convergence by using them for the first 4000 mini-batches that each device sees (out of potentially hundreds of thousands, depending on the training set). If I use local optimizers throughout the training, I achieve worse convergence results. I find the likely culprit for that is updating the local model shard, based on the gradient of only a single worker, uncoordinated with the rest of the shards, because it introduces extra noise during training.

Updating the parameter shard of the global model bears some resemblance to the Hogwild method (Recht et al., 2011) as I don't synchronize the updates to the shard, however, global updates are still synchronised. The concept of per-worker optimizers appears first in the work of Zhang et al. (2014), which served as an inspiration for my implementaion. Note that when applying delayed gradient updates, the effective batch size that the global optimizer deals with is τ times larger than the mini-batch size on which the local optimizers runs. Since local optimizers see gradients much smaller than the global optimizers, as they are a product of just one mini-batch, the learning rate they use is proportionally scaled down, compared to the one used by the global optimizers.

System	Hours	Epochs	BLEU	CE	CE T	WPS
optimized asynchronous (1)	4.97	10	35.56	50.90	18.72	40.2k
(1) + Aji and Heafield (2017)	4.32	11	35.16	52.02	19.66	41.5k
(1) + delayed updates $\tau = 2$ (2)	4.20	11	34.82	51.68	21.53	44.2k
(2) + local optimizers (3)	**3.66**	**10**	**35.45**	**51.32**	**18.77**	**44.2k**
(2) + warmup (4)	**4.87**	**13**	**35.29**	**50.78**	**19.36**	**44.2k**

Table 5.5: Romanian-English systems with different types of warmup. The systems in bold are the two new ones compared in this table. My local optimizers warmup system (3) significantly outperform learning rate warmup in terms of time-to-convergence.

I compare and contrast the two warmup strategies on Table 5.5. By itself, learning-rate warmup on top of delayed gradient updates offers slower convergence but to a better point compared to local optimizers at least in terms of cross-entropy. It does however take more epochs, which is perhaps because the initial updates were too small. Using local optimizers on the other hand reduces the number of epochs necessary to achieve convergence, improves the BLEU score on top just using delayed gradient updates and produces the best end-to-end training time thus far. Either of the warmup strategies result in lower training set cross-entropy. meaning that they both allow for the model to train better. I have shown that reducing communication in multi-GPU setting is the key to improving performance: My system with delayed gradient updates (2) achieves the fastest word-per-second rate. I also show that computation is much cheaper than inter-GPU communication, so when I introduce worker-local optimizers (system (3) on Table 5.5), there isn't any noticeable decrease in performance, despite the extra computation involved.

5.3.2 Momentum tuning and momentum cooldown

Goyal et al. (2017) and Vaswani et al. (2017) both employ cooldown strategies that lower the learning rate towards the end of training. Inspired by the work of Hadjis et al. (2016) however I decided to pursue a different cooldown strategy by modifying the momentum inside Adam's parameters (Kingma and Ba, 2015). Momentum is an addition to the standard SGD, usually computed as function of previous gradients which serves to smooth out updates, prevent spikes in gradients and push the model over saddle points.

Hadjis et al. (2016) show that fine tuning the momentum parameters is not a well

explored area in deep learning. Most researchers simply use the default values for momentum for a chosen optimizer. Hadjis et al. (2016) argue that this is an oversight especially when it comes to asynchronous SGD, because the asynchronicity adds extra implicit momentum to the training which is not accounted for. Because of this, asynchronous SGD has been deemed ineffective by Abadi et al. (2016), as without momentum tuning, the observed increase in training speed is negated by the lower convergence rate, resulting in near-zero net gain. I observed this as well: Training a model with delaying gradient updates and scaled learning rate requires more epochs to converge compared to my baseline system. However, Hadjis et al. (2016) show that after performing a grid search over momentum values, it is possible to achieve convergence rates typical for synchronous SGD even when working with many asynchronous workers. An example of the effect of momentum tuning can be seen on Figure 5.4 where reducing the momentum from 0.99 to 0.84 results in a much better convergence point in terms of cross-entropy as well as reduced number of steps towards convergence.

The downside of momentum tuning is that I can not offer rule-of-thumb values, as they are individually dependent on the optimizer used, the neural model, the number of workers and the batch size. Hadjis et al. (2016) perform grid search to substantially improve convergence rate and I take a similar route. In my experiments, I lowered the overall momentum in Adam using $\beta_1 = 0.93$ and $\beta_2 = 0.997$ (the default settings being $\beta_1 = 0.9$ and $\beta_2 = 0.999$). In addition to that I performed momentum cooldown where the momentum was further reduced to $\beta_1 = 0.92$ and $\beta_2 = 0.998$ after the first 4000 mini-batches that each worker sees. I identified these hyperparameters through a restricted grid search.

As we can see on Table 5.6, introducing momentum tuning on top of either of my warmup strategies results in improvements. The local optimizer system with momentum tuning (5) achieves better BLEU score and cross-entropy than just using local optimizers, while maintaining the same convergence time. The learning rate warmup system with momentum tuning (6) system needs fewer epochs to converge compared to just using learning rate warmup and also achieves better BLEU score and CE. Systems (5) and (6) do show worse training set cross-entropy compared to their non-momentum tuned versions, meaning those models are slightly more general, but the difference is quite small.
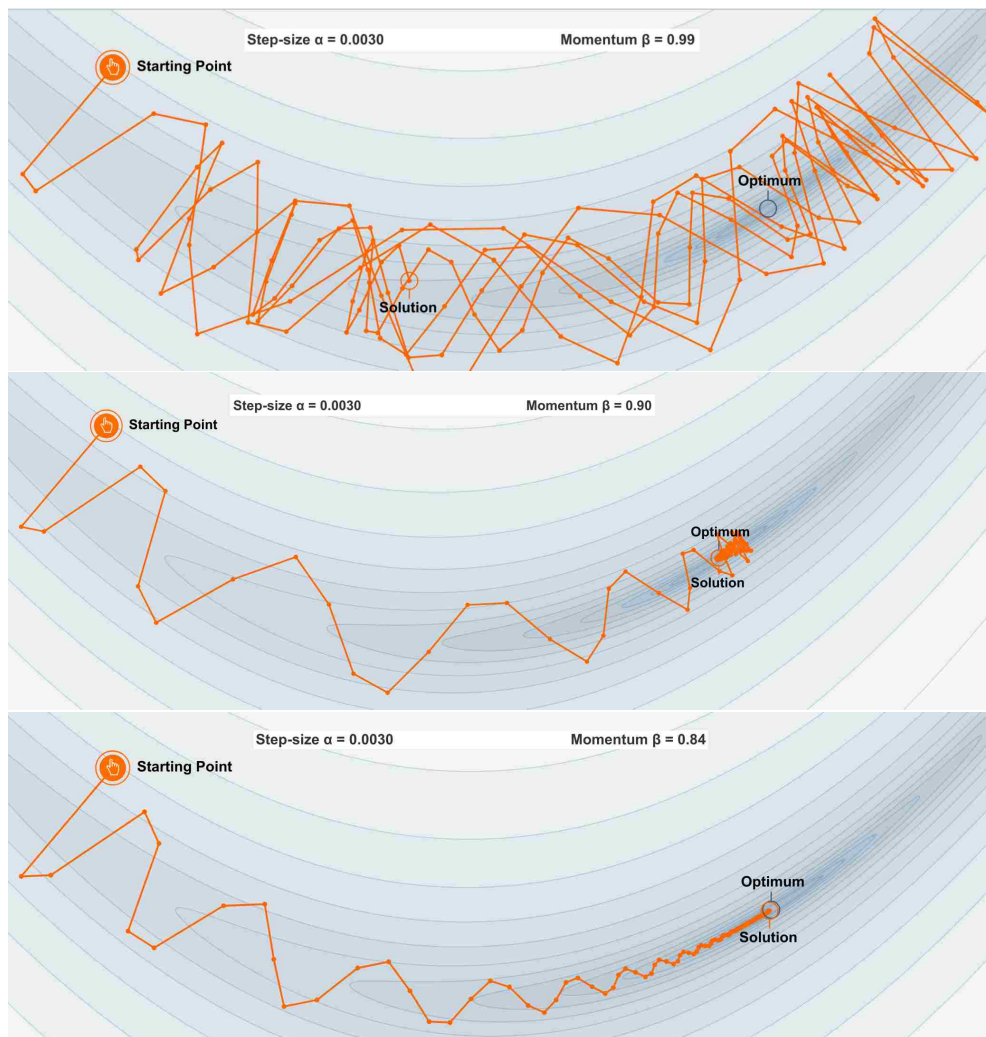
Figure 5.4: The effect of different momentum values can massively influence model convergence. The top image shows that when the momentum is too large, the model updates are too large, overshooting the desired change. The result is that many more updates are needed until the convergence point is reached. We can see how tuning the momentum gradually, results in fewer steps to converge on the middle image and the least on the bottom image. Image source: Goh (2017)

5.3.3 Results summary

Compared to the optimized baseline system (1), my best system (5) reduces the training time by 27%. Progression of the training can be seen on Figure 5.5. my system starts poorly compared to the baselines in terms of epoch-for-epoch convergence, but catches up in the later epochs. Due to faster training speed however, the desired BLEU score is achieved faster (Figure 5.5). To summarize the results:

- Delayed gradient updates improves training speed but worsens convergence, even when the learning rate is scaled with the increase of the mini-batch size.

System	Hours	Epochs	BLEU	CE	CE T	WPS
optimized asynchronous (1)	4.97	10	35.56	50.90	18.72	40.2k
(1) + Aji and Heafield (2017)	4.32	11	35.16	52.02	19.66	41.5k
(1) + delayed updates $\tau = 2$ (2)	4.20	11	34.82	51.68	21.53	44.2k
(2) + local optimizers (3)	3.66	10	35.45	51.32	18.77	44.2k
(2) + warmup (4)	4.87	13	35.29	50.78	19.36	44.2k
(3) + momentum tuning (5)	**3.66**	**10**	**35.48**	**50.87**	**18.80**	**44.2k**
(4) + momentum tuning (6)	**3.98**	**11**	**35.76**	**50.73**	**20.36**	**44.2k**

Table 5.6: Romanian-English results from my exploration and optimizations. The best results are achieved when I tune the momentum values on top of the previous best performing system. Once again the bolded text represents the newly compared systems in this table.

- Applying warmup helps convergence. I found that using local optimizers as a warmup strategy shows faster convergence compared to learning rate warmup at almost no penalty to BLEU or cross-entropy (System 5 vs system 6).

- Fine tuning momentum is the key to achieving best model convergence.

- Against the system equivalent to the one used in WMT 16 (Sennrich et al., 2016), my methods achieve nearly 4 times faster training time with no discernible penalty in BLEU or CE.

- For contrast, the other communication reducing method tested, the work of Aji and Heafield (2017), is slower than my work and achieves worse BLEU and CE.

5.4 Additional applications

I used the Romanian-English model for developing my methods, because it was fast to train, due to the training set being relatively small and the neural network being shallow. After I developed my methods and parameters, I tested them in several different settings, to confirm that the results generalize over a number of different settings:

5.4.1 A deep RNN

With deep RNN models, which contain multiple recurrent layers on both the encoder and decoder unlike shallow models, the parameters take more of the available VRAM
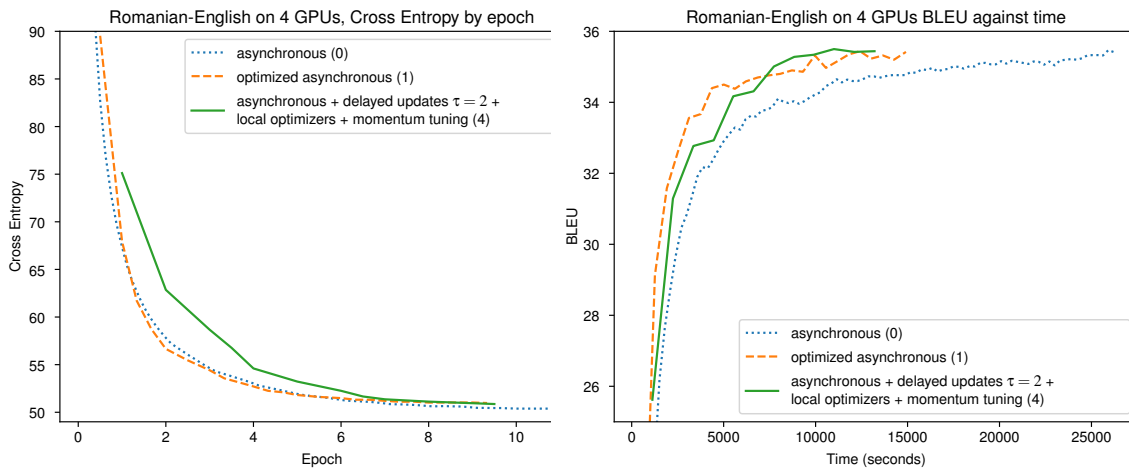
Figure 5.5: BLEU and cross-entropy training progression for the Romanian-English system. Due to the higher training speed and similar per-epoch training progression, my system achieves the target BLEU score faster.

leaving very little for mini-batches. I found that larger τ values are useful when dealing with such models. In this memory limited scenario, I apply $\tau = 4$ in order to achieve mini-batch size of similar magnitude as with the shallow model. I demonstrate the effectiveness of larger τ on Table 5.7.

System	Time (h)	BLEU	CE
Baseline	51.3	25.1	47.31
Async (5) + $\tau = 4$	39.7	25.07	46.59

Table 5.7: Training times for English-German deep RNN system trained on WMT17 data. My asynchronous system includes the optimizations of system (5) from Table 5.6.

The baseline system is equivalent to the second placed system for English-German at the WMT 2017 competition (Sennrich et al., 2017a). The baseline is trained with synchronous SGD and I use my best system from the Romanian-English experiments with the exception that I apply $\tau = 4$, instead of $\tau = 2$. I do not report the numbers for asynchronous baseline because I was unable to achieve competitive BLEU scores without using delayed gradient updates. I speculate this is because with this type of deep model, the mini-batch size is very small leading to very jerky and unstable training updates. Larger mini-batches ensure the gradients produced by different workers are going to be closer to one another. The training progression can be seen on Figure 5.6. I show that even though I use 4 times larger mini-batches, I actually manage to get lower Cross-Entropy epoch for epoch compared to the baseline (Figure 5.6). This,

coupled with higher training speed, allows my system to achieve 25.2 BLEU in 15 hours compared to 40 for the baseline system. (Figure 5.6).
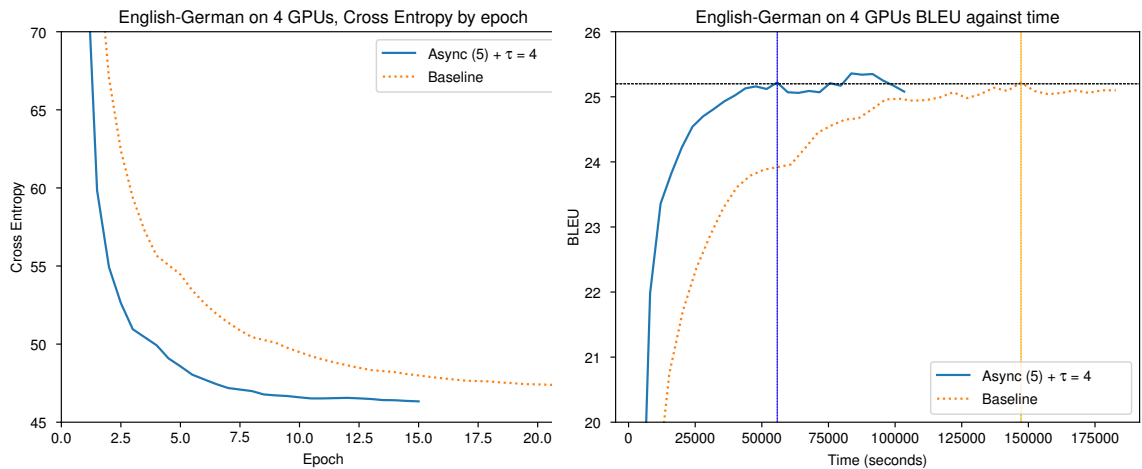

Figure 5.6: BLEU and cross-entropy training progression for the English-German system.

5.4.2 A high resource language pair

I built a French-English translation system in order to show how my methods perform on a high resource language pair. This model has the same hyperparameters as the Romanian-English system but it is trained on a lot more data: 35.9 Million parallel sentences taken from the WMT 15 datasets. Results are shown on Figure 5.7. The baseline system used is asynchronous SGD. The BLEU score progression is consistent with the CE progression. The system I developed achieves the maximum BLEU in 20 hours, compared to 33 hours for the baseline system.

5.4.3 A language model

I trained an RNN Language model on 57.8 Million English sentences collected from WMT 16 monolingual resources. I used a single layer GRU cell with 1024 dimension and 512 dimensions for the word embeddings. I am able to outperform the baseline system in terms of both cross-entropy and training time, converging in just half of the wall-clock time at a lower cross-entropy (Figure 5.8).
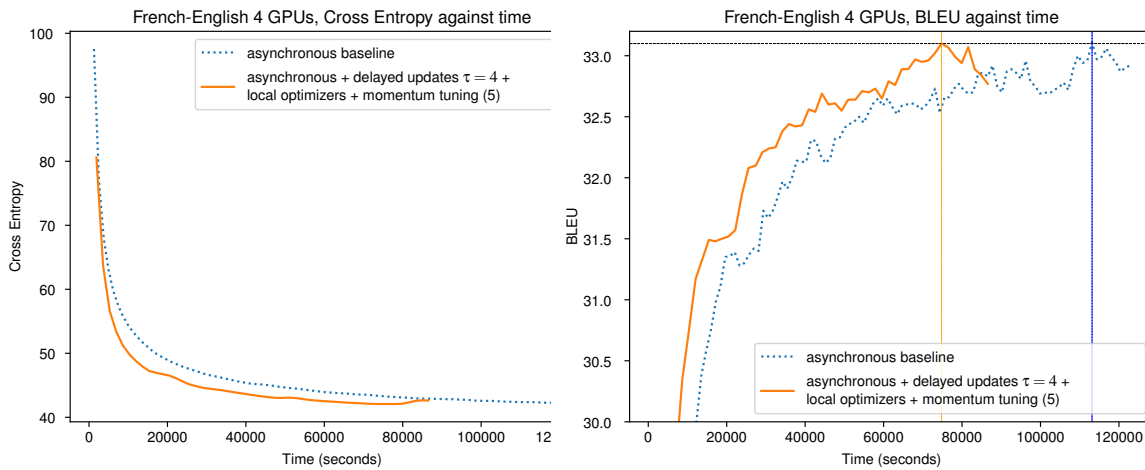
Figure 5.7: BLEU and cross-entropy training progression for the French-English system.
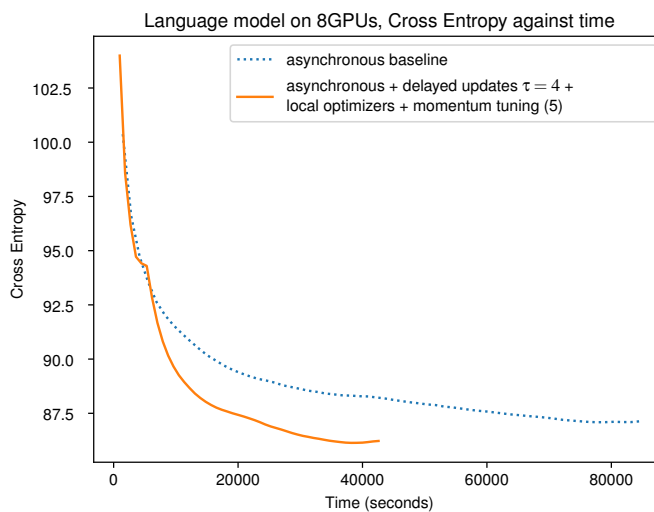
Figure 5.8: Language model training progression.

5.5 Training set Cross Entropy

In order to better understand the role that my modifications play on the model convergence, I plotted the training set cross entropy of the Romanian-English, French-English and the language modelling setting in the previous sections.

Figure 5.9 shows that the training set cross-entropy is quite similar on all systems measured. However due to the faster training speed, my improved system converges faster. Overall they follow a similar trend to the dev set results shown on Figure 5.5.

Figure 5.10 shows the cross-entropy training set progression. It is interesting to note that on the dev set (Figure 5.7) my best system is better than a baseline since the start of the training, however the same is not true on the training set. My system
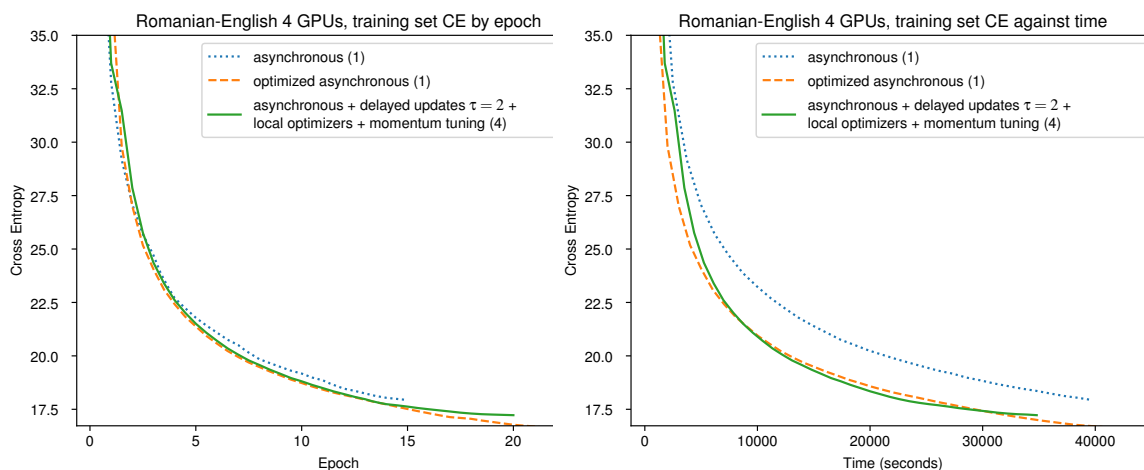
Figure 5.9: Training set cross entropy plots by epoch and against time on the Romanian-English systems from Figure 5.5.
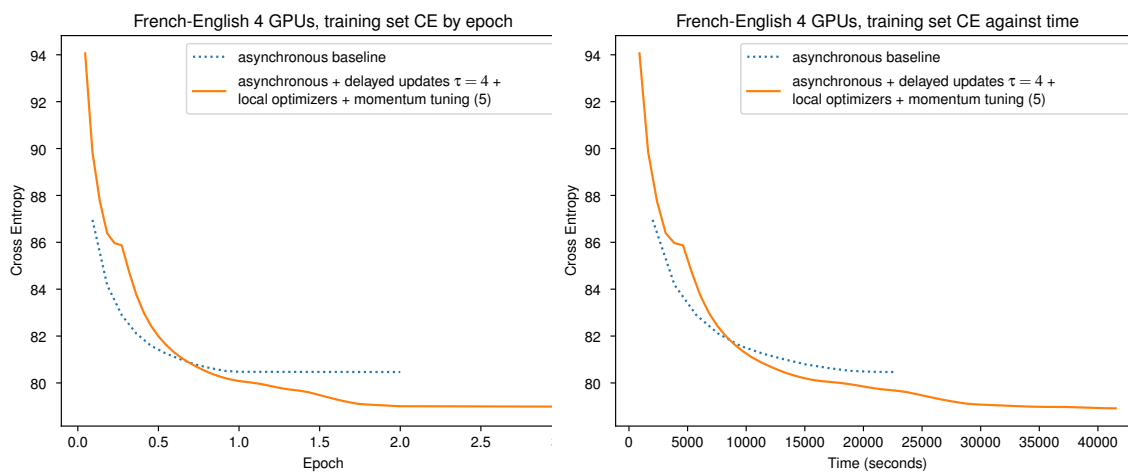
Figure 5.10: Training set cross entropy plots by epoch and against time on the French-English systems from Figure 5.7.

starts off slowly and only manages to overtake the baseline towards the end of the first training epoch after which it proceeds to arrive at a better convergence point compared to the baseline. The same observation is true for the language model experiment, shown on Figure 5.11.

After reviewing figures 5.9, 5.10, 5.11 I conclude that on the relatively small Romanian-English task my methods produce comparable cross-entropy results on the training set and any difference is likely the result of noise. On the French-English and language modelling tasks, which are much higher resource, it is clear that my methods improve the overall convergence of the model. Therefore any gains on BLEU or on cross-entropy on the dev set are the result of a better model convergence point, rather
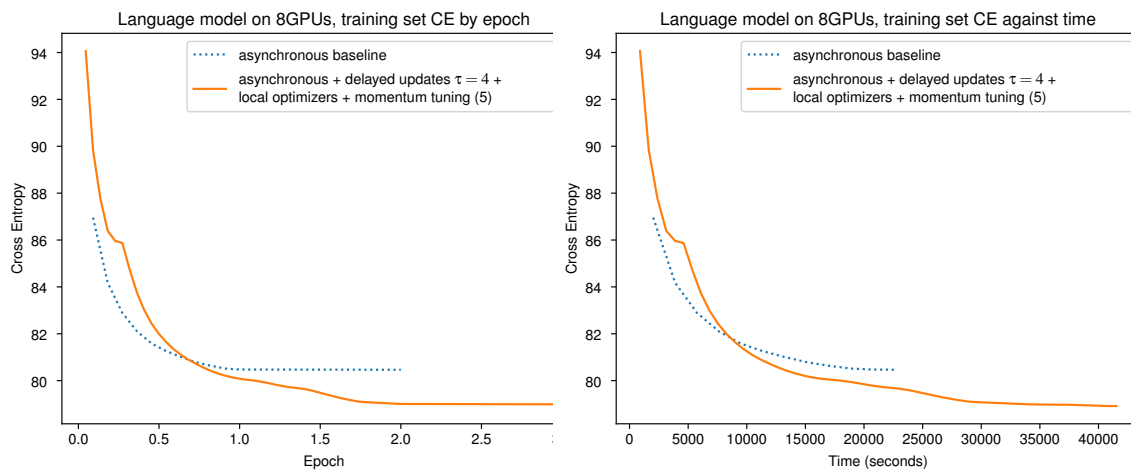
Figure 5.11: Training set cross entropy plots by epoch and against time on the language model systems from Figure 5.8.

than less overfitting towards the training set.

5.6 Further explorations

In this section I describe some methods that I developed but ultimately did not use in the best systems, because I found that they were very situational and did not otherwise generalize well.

5.6.1 Flexible learning rate

As described earlier, neural machine translation experiments have inherently imbalanced mini-batch sizes, because the sentences which form mini-batches are of varying length. To mitigate this neural machine translation systems commonly set a memory budget for mini-batches and try to fill in as many sentences as they can, grouping sentences of similar lengths together (Junczys-Dowmunt et al., 2018). This results in 'relatively' balanced mini-batch sizes in terms of number of words, even if internally they contain very different number of sentences. Even with this heuristic, there is significant variance between mini-batches which is only made worse by the introduction of delayed gradient updates (Table 5.8).

I hypothesized therefore that overall convergence could be improved by scaling the learning rate up or down, based on how much the mini-batch size of the particular mini-batch differs from the average mini-batch size. As shown on table 5.9, flexible learning rate does allow for slightly faster convergence, compared to my best system,

RAM	τ	AVG Words	σ Words
3 GB	1	2633	707
7 GB	1	5869	1594
13 GB	1	10566	3335
14* GB	2	11665	2439
26* GB	2	20955	4687
28* GB	4	23323	3655
56* GB	8	46379	4357

Table 5.8: Relationship between the RAM budget for batches (* means emulated by summing τ smaller batches) and number of source words processed in each batch. Both average and standard deviation σ are shown. Statistics were collected from 72 mini-batches with the asynchronous baseline system.

but in practice I found that the implementation complexity and the extra hyperparameter exploration would mean this feature is of little practical use.

System	Baseline		Flexible LR		Difference	
Romanian-English	hours	BLEU	hours	BLEU	hours	BLEU
Optimized asynchronous	4.46	35.57	4.48	35.19	+0.02	-0.38
+ $\tau = 2$	3.67	35.03	3.95	35.07	+0.28	+0.04
+ local optimizers & momentum tuning	3.66	35.48	**3.36**	**35.53**	**-0.3**	**+0.05**

Table 5.9: Effect of flexible learning rate application on top of the baselines and best systems of the Romanian-English model. When all optimizations are applied, flexible learning rate allows for slightly faster training time and a marginal improvement in BLEU.

5.6.2 Cyclic learning rate

I hypothesize that since it is difficult for us to know what is an appropriate learning rate for different stages of the training process, we could instead cycle continuously through learning rates during training. At the beginning of each cycle I drop the learning rate to a predefined minimum value and increase it linearly for a set number of batches until it reached the maximum learning rate. Then I lower it again to the predefined minimum and the next the cycle begins.

Additionally, I also reset the learning rate to the minimum value each time when there is no improvement in validation cost during a validation step. Empirically, I found that gradually increasing the learning rate and then dropping it over 40000 mini-batches works well. This is orthogonal to the usage of local optimizers. Similar ideas are expressed in the works of Loshchilov and Hutter (2016); Gotmare et al. (2018), who also search for different ways to find the optimal learning rate for any given training stage. Cyclical warmup did not provide any useful benefits for my test systems and the experimental results were quite inconclusive. They can be seen in Appendix A.

5.7 Related work

Work on stochastic gradient descent optimization has been quite popular over 2017-2018, with various work appearing that partially overlap with ours:

On reducing communication Inter-device communication is the limiting factor when scaling out SGD across multiple GPUs. Aji and Heafield (2017) address the issue by dropping the lower 99% of the gradient based on absolute value, thus substantially reducing the size of the communicated gradients, but at a slight cost of model quality. Mao et al. (2018) further refine the work of the Aji and Heafield (2017), addressing the model quality issue by applying warmup and momentum tuning.

On addressing momentum I point out earlier in this chapter, that it is possible the issues with convergence could be due to not accounting for the extra implicit momentum introduced by out methods. Independently from me, Shazeer and Stern (2018) have done further exploratory work on ADAM's momentum parameters using the transformer model (Vaswani et al., 2017) as a case study and have offered a mathematical explanation about why different stages of the training require different momentum values.

On delayed gradient updates Independently from me, Saunders et al. (2018) have employed delayed gradient updates in syntax NMT setting, where the sequences are much longer due to the syntax annotation and delayed updates are necessary because video RAM is limited.

On local optimizers Independently from me, Lin et al. (2018) have developed their own local optimizer solution as an alternative to increasing mini-batch sizes.

5.8   Conclusion and Future work

I show yet again that the key to achieving better training speed is to reduce communication, which in this case required a modification in asynchronous SGD. While I have demonstrated the methods on GPUs, they are hardware agnostic and can be applied to neural network training on any multi-device hardware such as TPUs or Xeon Phis. I was able to achieve up to three times faster end-to-end training on multiple tasks compared to the baseline systems. For the Romanian-English model, the training time nearly 3X shorter than the commonly used baseline and 1.5X shorter over a specifically optimised baseline. When experimenting with English-German the model is trained 1.3X faster than the baseline model, achieving practically the same BLEU score and one point lower cross-entropy score. A French-English system is trained 1.65X faster, achieving the same BLEU and better cross-entropy. My methods generalize well to other domains: Using them, I am able to train a language model 2X faster and to a much better convergence point.

In the future, I would like to apply local optimizers in distributed setting where the communication latency between local and remote devices varies significantly. In this setting, local optimizers may be used to synchronize remote models less often.

Chapter 6

Conclusion

In my dissertation I have taken a kaleidoscopic view of optimization centered around efficient memory accesses. I have examined three separate MT problems in the context of modern, parallel hardware and proposed modifications that improve performance. The common theme in all the cases is that big gains can be made from improving memory access patterns and reducing the communication between different computational units.

In chapter 3, I have introduced a phrase table for statistical machine translation that is used as the basis for a high performance phrase based decoder, delivering order-of-magnitude faster decoding speed than previous implementations. The performance improvements are due to improved memory locality and reducing the number of memory accesses during phrase query. This phrase table is later used inside a novel phrase based decoder designed for high performance.

In Chapter 4, I have introduced the first GPU-based n-gram language model, using novel data structures that compact and efficient with regards to memory accesses. While statistical machine translation research is out of fashion nowadays, it consistently outperforms neural machine translation in very low resource settings and is used in the heart of several unsupervised machine translation works (Artetxe et al., 2018; Lample et al., 2018). As the Internet and Facebook are breaking into the developing world, very low resource machine translation has become more important and this could be one place where the usage of phrase based machine translation would increase. I hypothesize that if there is again a sufficiently large demand for phrase based machine translation systems, my language model could be used as a basic building block for a phrase based decoder. In addition to that, my language model could be used independently in any setting where there is a need of large number of batch lan-

guage model queries, such as n-best-list rescoring or OCR.

In Chapter 5, I have introduced a method to accelerate neural network training in a multi-GPU setting, by reducing inter-GPU communication. Multi-GPU systems are increasingly becoming the norm, rather than the exception: 10-GPU machines are available for purchase commercially in late 2018 and computational clusters with hundreds of GPUs are widespread. We want to be able to distribute neural network training across as many GPUs and machines as possible and maintain close to linear scaling in order to not waste resources. My methods will potentially only become more popular over time and can be applied in principal to any neural network configuration or task, not just machine translation. This is a very hot research area, as evidenced by the large amount of published research over the past year (Aji and Heafield, 2017; Mao et al., 2018; Shazeer and Stern, 2018; Lin et al., 2018; Saunders et al., 2018; Loshchilov and Hutter, 2016; Gotmare et al., 2018).

Potential impacts of optimization

Cost   By optimizing for speed we can substantially decrease the hardware cost necessary for conducting large scale NLP. In chapter 4, I show that in a language modeling task, a specially crafted language model is twice more efficient per £ for hardware compared to traditional CPU implementations.

Support for modern hardware   The work I have done focuses primarily on parallel, massively parallel and distributed[1] hardware. As discussed in Chapter 2, with the evolution of computer hardware, there are diminishing gains from adding extra transistors on a single chip so hardware manufacturers have focused on multi-chip or multi-device solutions. In Chapter 3, I show how a phrase table for statistical machine translation designed for modern multi-core hardware can outperform previous solutions by an order of magnitude. In Chapter 4, I show how *n*-gram language modeling could take advantage of a GPU, a hardware with higher theoretical performance than a CPU. In Chapter 5, I show that reducing communication is key in order to take advantage of multi-GPU hardware for training neural machine translation models.

Quality improvement   Gustafson's law (Gustafson, 1988) points out that improving performance means that more resources can be allocated to a given task and thus pro-

---

[1]Multi-device is a type of distributed

duce a better result. Due to improved phrase Table and language model performance we can increase the exploration of the search space and thus achieve better translations or simply decrease the time necessary for a single experiment and allow researchers to perform more in-depth hyperparameter exploration. In the neural machine translation setting, where it is frequently impossible to train a model to convergence due to time constraints, improved performance means we can use larger models with more parameters, or smaller models to a better convergence point.

Multi-core systems will not scale forever   Empirical evidence suggests that we will eventually hit a wall in the scaling of multi-core processors, because of an effect known as Dark Silicon (Esmaeilzadeh et al., 2011). Dark silicon is an effect where as transistors get smaller, they can no longer be powered at their nominal TDP. This is a completely a breakdown of Dennard scaling (Dennard et al., 1974), which states that as transistors go smaller, their power density stays constant. According to Esmaeilzadeh et al. (2011), at the 8nm level, dark silicon may account for 60-80% of the chip transistors, meaning that the heat produced, when enough power is provided to power all the transistors on the chip, will be way above the safe operating ranges. For a reference point, Intel is expected to introduce their first 10nm chips in 2019 or 2020. This means that in order to continue with performance improvements, we need to make best use of the hardware available, and in the near future, distributed computing is likely to become more prominent, as it deals with the heat dissipation issues. In the latter case specifically, the key to good performance is the ability to make efficient use of memory and limit communication.

In the era of big data, there is increasingly more data available for variety of NLP tasks. We can not feasibly process all available data if we just rely on the incremental performance improvements delivered by the annual hardware refreshes: the techniques and ideas in the thesis will contribute to more efficient use of the available hardware.

Appendix A

Cyclical warmup and random learning rate

When I tried $\tau = 4$ over $\tau = 2$ I achieved 7k faster WPS processing on 4 GPUs. When I tried the same with 8 GPUs, I got 10k faster WPS, which is 30% higher. I wanted to use the extra speedup to achieve even shorter end-to-end training time, however I was unable to maintain equivalent model convergence in terms of epochs, which is contrary to the goals I set up earlier: Faster training speed without a regression in model quality or epochs to convergence. Figure A.1 shows my results. I could not get $\tau = 4$ to work on the shallow Romanian-English model even though it works well for the deep English-German model. Cyclical learning rate warmup does help a bit with the convergence, but not enough for it to be a viable strategy.

GPUs	Language pair	τ	Cyclical warmup	WPS	Epochs to converge
4	ro-en	2	no	40.2k	10
4	ro-en	4	no	47.6k	16
8	ro-en	2	no	62.7k	10
8	ro-en	4	no	73.2k	24
8	ro-en	4	yes	73.2k	23

Table A.1: Cyclical warmup does improve the convergence when using larger τ but even so we are unable to maintain the same convergence rate in terms of epochs compared as using $\tau = 2$. I do not report end-to-end training times, because different types of GPUs were used for some of the experiments, which makes the metric inaccurate. Epochs to converge on the other hand is universal across models, provided they use the same hyperparameters.

After my experiments with cyclical learning rate, I attempted to use completely random learning rate for each mini-batch, with the random number drawn from an uniform distribution $[0, \tau * 0.00045]$, ($0.00045 * \tau$ is the learning rate I used in the rest of the experiments), but this did not help with convergence. Surprisingly, it did not make convergence much worse either.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zhang, X. (2016). Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695.

Aji, A. F. and Heafield, K. (2017). Sparse communication for distributed gradient descent. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark.

Andalam, S., Sinha, R., Roop, P., Girault, A., and Reineke, J. (2013). Precise modelling of instruction cache behaviour.

Aoe, J., Morimoto, K., and Sato, T. (1992). An efficient implementation of trie structures.

Argueta, A. and Chiang, D. (2017). Decoding with finite-state transducers on gpus. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1044–1052. Association for Computational Linguistics.

Argueta, A. and Chiang, D. (2018). Composing finite state transducers on gpus. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 2696–2704.

Artetxe, M., Labaka, G., and Agirre, E. (2018). Unsupervised statistical machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3632–3642.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. In *ICLR 2015*.

Bayer, R. and McCreight, E. (1970). Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA. ACM.

Bogoychev, N. and Hoang, H. (2016). Fast and highly parallelizable phrase table for statistical machine translation. In *Proceedings of the First Conference on Machine Translation*, pages 102–109, Berlin, Germany. Association for Computational Linguistics.

Bogoychev, N., Junczys-Dowmunt, M., Heafield, K., and Aji, A. F. (2018). Accelerating asynchronous stochastic gradient descent for neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium.

Bogoychev, N. and Lopez, A. (2016). N-gram language models for massively parallel devices. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Bojar, O., Chatterjee, R., Federmann, C., Graham, Y., Haddow, B., Huck, M., Jimeno Yepes, A., Koehn, P., Logacheva, V., Monz, C., Negri, M., Neveol, A., Neves, M., Popel, M., Post, M., Rubino, R., Scarton, C., Specia, L., Turchi, M., Verspoor, K., and Zampieri, M. (2016). Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation*, pages 131–198, Berlin, Germany. Association for Computational Linguistics.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *In Proceedings of EMNLP-CoNLL*.

Callison-Burch, C., Bannard, C., and Schroeder, J. (2005). Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 255–262. Association for Computational Linguistics.

Canny, J., Hall, D., and Klein, D. (2013). A multi-teraflop constituency parser using GPUs. In *Proc. of EMNLP*.

Carvalho, C. (2002). The gap between processor and memory speeds.

Cesarini, F. and Soda, G. (1983). An algorithm to construct a compact b-tree in case of ordered keys. *Information Processing Letters*, 17(1):13–16.

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–393.

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

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*.

Chong, J., Gonina, E., Yi, Y., and Keutzer, K. (2009). A fully data parallel wfstbased large vocabulary continuous speech recognition on a graphics processing unit. In *in 10th Annual Conference of the International Speech Communication Association (InterSpeech*.

Chong, J., Yi, Y., Faria, A., Satish, N. R., and Keutzer, K. (2008). Data-parallel large vocabulary continuous speech recognition on graphics processors. Technical Report UCB/EECS-2008-69, EECS Department, University of California, Berkeley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. (2012). Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Dennard, R. H., Gaensslen, F. H., nien Yu, H., Rideout, V. L., Bassous, E., Andre, and Leblanc, R. (1974). Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA. ACM.

Federico, M., Bertoldi, N., and Cettolo, M. (2008). Irstlm: an open source toolkit for handling large scale language models. In *INTERSPEECH*, pages 1618–1621. ISCA.

Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.

Gage, P. (1994). A new algorithm for data compression. *C Users J.*, 12(2):23–38.

Ge, R., Huang, F., Jin, C., and Yuan, Y. (2015). Escaping from saddle points — online stochastic gradient for tensor decomposition. In Grünwald, P., Hazan, E., and Kale, S., editors, *Proceedings of The 28th Conference on Learning Theory*, volume 40 of *Proceedings of Machine Learning Research*, pages 797–842, Paris, France. PMLR.

Germann, U. (2014). Dynamic phrase tables for machine translation in an interactive post-editing scenario. In *Proceedings of the Workshop on Interactive and Adaptive Machine Translation*, pages 20–31.

Germann, U. (2015). Sampling phrase tables for the moses statistical machine translation system. *Prague Bull. Math. Linguistics*, 104:39–50.

Goh, G. (2017). Why momentum really works. *Distill*.

Gotmare, A., Keskar, N. S., Xiong, C., and Socher, R. (2018). A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. *CoRR*, abs/1810.13243.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677.

Green, S., Cer, D., and Manning, C. (2014). Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of WMT*.

Gülçehre, Ç., Firat, O., Xu, K., Cho, K., Barrault, L., Lin, H., Bougares, F., Schwenk, H., and Bengio, Y. (2015). On using monolingual corpora in neural machine translation. *CoRR*, abs/1503.03535.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533.

Haddow, B., Bogoychev, N., Emelin, D., Germann, U., Grundkiewicz, R., Heafield, K., Miceli Barone, A. V., and Sennrich, R. (2018). The University of Edinburgh's Submissions to the WMT18 News Translation Task. In *WMT 2018*, Brussels, Belgium. Association for Computational Linguistics.

Hadjis, S., Zhang, C., Mitliagkas, I., and Re, C. (2016). Omnivore: An optimizer for multi-device deep learning on cpus and gpus.

Hall, D., Berg-Kirkpatrick, T., and Klein, D. (2014). Sparser, better, faster GPU parsing. In *Proc. of ACL*.

He, H., Lin, J., and Lopez, A. (2013). Massively Parallel Suffix Array Queries and On-Demand Phrase Extraction for Statistical Machine Translation Using GPUs. In *HLT-NAACL*, pages 325–334. The Association for Computational Linguistics.

He, H., Lin, J., and Lopez, A. (2015). Gappy Pattern Matching on GPUs for On-Demand Extraction of Hierarchical Translation Grammars. *TACL*, 3:87–100.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society.

Heafield, K. (2011). KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, United Kingdom.

Heafield, K. (2013). *Efficient Language Modeling Algorithms with Applications to Statistical Machine Translation*. PhD thesis, Carnegie Mellon University.

Heafield, K., Kshirsagar, R., and Barona, S. (2015). Language identification and modeling in specialized hardware. In *Proceedings of ACL-IJCNLP*.

Hennessy, J. L. and Patterson, D. A. (2018). *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition.

Hoang, H., Bogoychev, N., Schwartz, L., and Junczys-Dowmunt, M. (2016). Fast, scalable phrase-based SMT decoding. *CoRR*, abs/1610.04265.

Hwu, W.-m. W. (2011). *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Intel Corporation (2009). *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Intel Corporation.

Junczys-Dowmunt, M. (2012). Phrasal rank-encoding: Exploiting phrase redundancy and translational relations for phrase table compression. *Prague Bull. Math. Linguistics*, 98:63–74.

Junczys-Dowmunt, M. (2012). A space-efficient phrase table implementation using minimal perfect hash functions. In *Text, Speech and Dialogue - 15th International Conference, TSD 2012, Brno, Czech Republic, September 3-7, 2012. Proceedings*, pages 320–327.

Junczys-Dowmunt, M., Grundkiewicz, R., Dwojak, T., Hoang, H., Heafield, K., Neckermann, T., Seide, F., Germann, U., Fikri Aji, A., Bogoychev, N., Martins, A. F. T., and Birch, A. (2018). Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations*, pages 116–121, Melbourne, Australia. Association for Computational Linguistics.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR*.

Koehn, P. (2010). *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., Dyer, C., Bojar, O., Constantin, A., and Herbst, E. (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*, ACL '07, pages 177–180, Stroudsburg, PA, USA. Association for Computational Linguistics.

Kudo, T., Hanaoka, T., Mukai, J., Tabata, Y., and Komatsu, H. (2011). Efficient dictionary and language model compression for input method editors. In *Proceedings of*

the Workshop on Advances in Text Input Methods, WTIM@IJCNLP 2011, Chiang Mai, Thailand, November 13, 2011, pages 19–25.

Lample, G., Ott, M., Conneau, A., Denoyer, L., and Ranzato, M. (2018). Phrase-based & neural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5039–5049. Association for Computational Linguistics.

Lin, T., Stich, S. U., and Jaggi, M. (2018). Don't use large mini-batches, use local sgd. *CoRR*, abs/1808.07217.

Lopez, A. (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)*, pages 976–985, Prague, Czech Republic. Association for Computational Linguistics.

Lopez, A. (2008). *Tera-Scale Translation Models via Pattern Matching*, pages 505–512. Coling 2008 Organizing Committee.

Loshchilov, I. and Hutter, F. (2016). SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983.

Mao, H., Han, S., Yu, W., Dally, W., and Lin, Y. (2018). Deep gradient compression: Reducing the communication bandwidth for distributed training.

Masters, D. and Luschi, C. (2018). Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612.

McFarland, G., Mcfarl, G., Flynn, M., and Flynn, M. (1995). Limits of scaling mosfets.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Neubig, G. and Dyer, C. (2016). Generalizing and hybridizing count-based and neural language models. In *EMNLP*, pages 1163–1172. The Association for Computational Linguistics.

Nick Cercone, Max Krause, J. B. (1983). Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *CAMWA*, 9(1):215–231.

NVIDIA Corporation (2015). *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, New York, NY, USA. ACM.

Osborne, M., Lall, A., and Durme, B. V. (2014). Exponential reservoir sampling for streaming language models. In *Proceedings of ACL*, pages 687–692.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA. Association for Computational Linguistics.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). A case for intelligent ram. *IEEE Micro*, 17(2):34–44.

Perrone, M. (2009). Multicore programming challenges. In *European Conference on Parallel Processing*, pages 1–2. Springer.

Peterson, W. W. (1957). Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc.

Rosenberg, A. L. and Snyder, L. (1981). Time- and space-optimality in b-trees. *ACM Trans. Database Syst.*, 6(1):174–193.

Saunders, D., Stahlberg, F., de Gispert, A., and Byrne, B. (2018). Multi-representation ensembles and delayed sgd updates improve syntax-based nmt. *CoRR*, abs/1805.00456.

Schweizer, H., Besta, M., and Hoefler, T. (2015). Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456.

Sennrich, R., Birch, A., Currey, A., Germann, U., Haddow, B., Heafield, K., Miceli Barone, A. V., and Williams, P. (2017a). The University of Edinburgh's Neural MT Systems for WMT17. In *Proceedings of the Second Conference on Machine Translation, Volume 2: Shared Task Papers*, Copenhagen, Denmark.

Sennrich, R., Firat, O., Cho, K., Birch, A., Haddow, B., Hitschler, J., Junczys-Dowmunt, M., Läubli, S., Miceli Barone, A. V., Mokry, J., and Nadejde, M. (2017b). Nematus: a Toolkit for Neural Machine Translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 65–68, Valencia, Spain. Association for Computational Linguistics.

Sennrich, R., Haddow, B., and Birch, A. (2016). Edinburgh Neural Machine Translation Systems for WMT 16. In *Proceedings of the First Conference on Machine Translation*, pages 371–376, Berlin, Germany. Association for Computational Linguistics.

Shazeer, N. and Stern, M. (2018). Adafactor: Adaptive learning rates with sublinear memory cost. *CoRR*, abs/1804.04235.

Talbot, D. and Osborne, M. (2007). Smoothed Bloom Filter Language Models: Tera-Scale LMs on the Cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 468–476, Prague, Czech Republic. Association for Computational Linguistics.

Tillmann, C. and Ney, H. (2003). Word reordering and a dynamic programming beam search algorithm for statistical machine translation. *Comput. Linguist.*, 29(1):97–133.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Watanabe, T., Tsukada, H., and Isozaki, H. (2009). A succinct N-gram language model. In *Proc. of ACL-IJCNLP*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser,

Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

Yasuhara, M., Tanaka, T., ya Norimatsu, J., and Yamamoto, M. (2013). An efficient language model using double-array structures. In *EMNLP*, pages 222–232. ACL.

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press.

Zens, R. and Ney, H. (2007). Efficient phrase-table representation for machine translation with applications to online mt and speech translation. In *Human Language Technology Conf. / North American Chapter of the Assoc. for Computational Linguistics Annual Meeting*, pages 492–499, Rochester, NY.

Zhang, S., Choromanska, A., and LeCun, Y. (2014). Deep learning with elastic averaging SGD. *CoRR*, abs/1412.6651.

Zhang, Y. and Vogel, S. (2005). An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *In Proceedings of the 10th Conference of the European Association for Machine Translation (EAMT-05*, pages 30–31.