Parallelisation of Text Mining Algorithms

Third Year Project Report

Student:  Todd Davies
Supervisor:  Dr. Richard Neville
Degree Programme:  BSc (Hons) Computer Science
Year:  2016
# Contents

1 Introduction
   1.1 Chapter Overview .................................................. 5
   1.2 The Problem ......................................................... 5
   1.3 Report Structure .................................................... 6
   1.4 Requirements ....................................................... 6
   1.5 Previous Work ....................................................... 7
   1.6 Chapter Conclusion ................................................ 7

2 Part of Speech Tagging
   2.1 Chapter Overview .................................................... 8
   2.2 Defining the Problem .............................................. 8
   2.3 A Comparison of Part of Speech Tagging Algorithms .......... 9
      2.3.1 Frequency Based Model ....................................... 10
      2.3.2 Hidden Markov Model ....................................... 10
      2.3.3 Rule Based Model .......................................... 11
   2.4 Choosing a Model .................................................. 11
   2.5 Brill Tagging ....................................................... 11
      2.5.1 Algorithm Overview ........................................ 12
      2.5.2 Applying the Model ....................................... 12
      2.5.3 Training the model ....................................... 13
   2.6 Chapter Conclusion ................................................ 15

3 Parallelisation
   3.1 Chapter Overview .................................................. 16
   3.2 Background ......................................................... 16
   3.3 Parallelisation on a Single Thread ............................ 17
   3.4 Parallelisation with CPU Threads ............................... 18
      3.4.1 Parallel Patterns Library ................................ 18
   3.5 GPU Accelerated Computation .................................. 19
      3.5.1 Design Goals of the GPU ................................ 19
      3.5.2 Architecture of the GPU .................................. 20
      3.5.3 CUDA ........................................................ 21
   3.6 Chapter Conclusion ................................................ 23

4 Experiments and Analysis
   4.1 Chapter Overview .................................................. 24
   4.2 Experimental Design and Methodology ......................... 24
      4.2.1 Programming Language ..................................... 24
      4.2.2 Development Tools ......................................... 25
      4.2.3 Testing ....................................................... 26
   4.3 Experiment 1: Parallelising Computation of the Collatz Conjecture ........................................... 26
   4.4 Experiment 2: Parallelising the Scoring of Brill Rules .... 26
Abstract

In this report, parallel implementations of training algorithms for part of speech tagging models are introduced, targeting both conventional CPU based methods of parallelism and massively parallel General Purpose GPU (GPGPU) computation.

These techniques are applied to training a model for Brill’s tagging algorithm, where a performance increase of between 13x and 100x is obtained on commodity GPU hardware, found in many desktop computers today.

In particular, the intricacies of GPGPU computation are discussed, along with its benefits and drawbacks. Techniques are described that can be used to map irregularly sized text processing problems onto platforms that are optimised for efficient computation of operations over large, regularly-sized datasets.

This work has applications in the field of text mining, where Machine Learning (ML) models are often used to extract high quality information from bodies of text. Training these models can take significant time and resources, therefore approaches that parallelise the computation in order to fully utilise the available hardware are of particular interest.
Chapter 1

Introduction

1.1 Chapter Overview

This chapter introduces and explains what Text Mining (TM) is, and details the scope of the project as well as its motivating factors. The overall structure of this document is then given, before the project requirements are explicated. Finally an outline of the previous work done by other authors on this topic is presented.

1.2 The Problem

Text mining is used by many different industries to analyse large datasets, automatically deriving information that could take weeks or months to process manually. Example applications for TM include processing patient records in healthcare, analysing academic papers and validating tax returns; in short, most industries with vast amounts of data are able to employ and benefit from TM techniques.

In order for a computer to analyse a document, a number of steps must be performed to transform the input data from natural language into a machine-understandable representation, termed a text mining pipeline. Different TM applications have different pipelines according to their needs; one may be to cluster together similar documents from an online library, or another trying to extract the structure of proteins described by PhD theses on drug discovery.

Due to their heterogeneity, different TM pipelines are composed of different functional steps, but in the general case, the input text is first tokenized and then each token is tagged with a part of speech (POS). While, tokenization is a trivial task that can often be computed in linear time, POS tagging is a more complex problem, which requires the use of computationally expensive machine learning (ML) approaches to solve with acceptable results.

---

A natural language is some human language such as English, with diverse, flexible and context dependent rules. In contrast a formal (machine-understandable) language is one with a more regular structure, that can be fully and unambiguously understood with a far smaller set of rules.

Tokenization (in the context of lexical analysis), is where input text is split into tokens, such that each token corresponds to some meaningful element of text such as a word, part of a word, or a punctuation mark.

This process is described in more detail in Chapter 2.

Identifying word boundaries and punctuation in text is a simple task for most languages such as English.
With a view to making POS tagging as accurate as possible, machine learning models are often trained on data specific to the domain in which they will be operating; for example, a tagger trained on data from academic papers would almost certainly get a poor accuracy if it was used to tag text from Twitter user’s tweets. Furthermore, different models must be produced for different languages, and must be re-trained as the language changes over time. This means that training a tagger is not as rare an event as one might imagine.

To create an accurate model, the training algorithm must be given data that represents a wide variety of real-world scenarios so that as many linguistic quirks and edge cases are encapsulated in it as possible. Training with the full amount of available data may not be feasible due to time constraints, leading to a subset of the training data being used instead, which may result in a sub-optimal model.

The aim of this project is to research and investigate how the performance of the training stage for POS tagging applications can be enhanced, so that more data can be used in training a model, as to improve its accuracy. Improving the performance of an early stage of the TM pipeline increases the number of TM applications that this work is applicable to, due to most sharing common processing stages early on.

### 1.3 Report Structure

This report is structured as to first set the scene by describing the problem and its proposed solution (Chapter 1), before going on to introduce key background information about POS tagging (Chapter 2) and parallelisation (Chapter 3). The design and implementation of the experiments performed during the course of the project are described (Chapter 4), and finally, a summary of the its achievements of is presented, along with suggestions for further work that could be done on the topic (Chapter 5).

### 1.4 Requirements

A set of functional and non-functional requirements were formulated (given in Table 1.1a and Table 1.1b) to reflect the goals of the project and to provide a measure against which the outcome of the project can be evaluated.

---

5Languages evolve over time, and will adapt to the needs of whoever is using them. Use of language on social media can be especially fluid as both slang and abbreviations are commonly used, where specific phrases come in and out of vogue within short time periods.
Table 1.1: Project Requirements

(a) Functional Requirements

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>Find a corpus to use for testing</td>
<td>High</td>
</tr>
<tr>
<td>FR2</td>
<td>Investigate POS tagging algorithms</td>
<td>High</td>
</tr>
<tr>
<td>FR3</td>
<td>Study and implement a widely used POS tagger</td>
<td>High</td>
</tr>
<tr>
<td>FR4</td>
<td>Investigate parallelisation methods for CPU’s</td>
<td>Medium</td>
</tr>
<tr>
<td>FR5</td>
<td>Investigate parallelisation methods for GPGPU computation</td>
<td>High</td>
</tr>
<tr>
<td>FR6</td>
<td>Parallelise POS tagging training on a CPU</td>
<td>Medium</td>
</tr>
<tr>
<td>FR7</td>
<td>Parallelise POS tagging training on a GPU</td>
<td>High</td>
</tr>
<tr>
<td>FR8</td>
<td>Benchmark the results of the three implementations</td>
<td>High</td>
</tr>
<tr>
<td>FR9</td>
<td>Analyse the results of the benchmark</td>
<td>High</td>
</tr>
</tbody>
</table>

(b) Non-Functional Requirements

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR1</td>
<td>Implementations should be corpora agnostic and work with any sensibly formatted input data</td>
<td>Medium</td>
</tr>
<tr>
<td>NFR2</td>
<td>Implementations should be able to use any set of POS tags</td>
<td>Medium</td>
</tr>
<tr>
<td>NFR3</td>
<td>Different implementations of the same POS tagging algorithm should get the same results</td>
<td>High</td>
</tr>
<tr>
<td>NFR4</td>
<td>Different implementations of the same POS tagger should be directly comparable for performance</td>
<td>High</td>
</tr>
</tbody>
</table>

The requirements of the project are categorised as being either functional or non-functional according to their characteristics. Each requirement was also given a priority to reflect how critical it was to the success of the project, and to help indicate which requirements should be given precedence at implementation time.

1.5 Previous Work

With the advent of massively parallel multi-core systems coinciding with an explosion of interest in distributed computing, the former has been given comparatively little attention by academics aiming to increase the performance of TM applications, though some work has been done on parts of the TM pipeline other than POS tagging [1]. Sharing computational load between multiple machines in a data centre is a common approach to parallelisation [2], since frameworks such as Hadoop\(^6\) and OpenMPI\(^7\) are very good at abstracting the complicated details of distributed computation away from the programmer. In contrast, this project aims to maximise the performance of a single machine, and is envisaged as being useful when a compute cluster is not available, or for fully utilising the resources of machines in an existing cluster.

1.6 Chapter Conclusion

This chapter introduced the motivating factors behind the project as well as outlining its requirements. The structure of the report was described, and a summary of previous work in the literature was given.

---


\(^7\) [https://www.open-mpi.org/](https://www.open-mpi.org/)
Chapter 2

Part of Speech Tagging

2.1 Chapter Overview

This chapter gives background information about POS tagging, details widely used POS tagging algorithms, and elucidates the reasoning behind the decision of selecting which ones to parallelise during the course of the project.

2.2 Defining the Problem

All words belong to a part of speech, determined by both the semantic functions performed by that word and the context in which the word appears. After the input text has been tokenized, the next step in many TM pipelines is to impose more structure upon it by assigning each token a POS tag, as illustrated in Figure 2.1.

![Figure 2.1: The architecture of the start of a TM pipeline. Input text is fed to a tokenizer, which splits it into tokens, and passes it into a POS tagger. Each token is then assigned a tag in order to assist in whatever analysis is performed by later steps in the pipeline.](image)

---

8 Such as a noun, verb, adjective, etc.
9 For example, the words ‘Manchester’ and ‘London’ are proper nouns since they are used to identify a place.
10 The context of a word is defined as the parts of a statement preceding and following the word, which influences the meaning of the word.
Figure 2.2: Here, the first portion of two different sentences containing the word ‘start’ is shown. In Figure 2.2a, ‘start’ appears as a verb, but in Figure 2.2b, it appears as a noun. This serves as a motivating example showing how POS tagging is dependent not only on the word itself, but also on the surrounding words that make up its context.

Throughout the project, the British National Corpus (BNC) was used as the test data for all experiments primarily due to its large size, but also since it contains material from a wide range of sources. This decision dictated the exact set of POS tags used in the project, since the BNC uses the C5 tagset, a complete listing of which is given in Appendix B.1.

However, the complexity of POS tagging has little to do with the size of the employed dataset or tagset, but instead arises from the fact that the tag given to a word is often dictated by the context of the word. For example, the word ‘start’ is commonly assigned two different parts of speech tags; verb and noun, as illustrated by Figure 2.2.

The context of a word isn’t the only factor that must be considered when tagging; for example, if the word is a homograph, then the decision of what POS tag to assign is complicated further, since words with the same spellings can have different meanings and different effects on the sentence to which they belong.

Historically, POS tagging was performed by professional linguists, but such a repetitive task is an obvious candidate for automation in order to reduce the amount of labour required and speed up the process. As such, a number of POS tagging algorithms have been conceived, many of which are outlined and considered in the next section.

2.3 A Comparison of Part of Speech Tagging Algorithms

During the research phase of the project, the author considered seven different tagging algorithms. This section describes three approaches; frequency based tagging, hidden markov models, and rule based tagging, while descriptions of other less relevant algorithms are given in Appendix A.1. The suitability of each algorithm as a candidate for parallelisation is also analysed and discussed.

All of the considered algorithms employ some machine learning techniques in order to use training data to make predictions about previously unseen data. As was depicted in Figure 2.1, machine learning typically involves two stages; training and testing.

---

11 A collection of 100 million POS tagged words from a variety of sources [3], including recordings of every day conversations, newspaper articles and television reports.

12 A tagset is the set of tags that are assigned to tokens in a corpus. Each tag is a three letter code corresponding to some part of speech, such as ‘VVB’, which denotes a verb.

13 Two or more words with the same spelling that have different meanings are referred to as homographs. For example, the word ‘bank’ can either refer to a financial institution, the side of a river, a store of some resource (such as a seed bank), or an aerobatic manoeuvre!
2.3.1 Frequency Based Model

A frequency based model analyses large amounts of tagged training data, and extracts statistics indicating the most common POS tag for each token that was present. The algorithm constructs a mapping from tokens to POS tags during the training phase, so that when a token is to be assigned a tag in the testing phase, it is simply found in the mapping and the relevant POS tag is returned. An example of this is shown in Figure 2.3.

Frequency models are easy to understand and implement, exhibiting both linear time training and tagging performance. However, this simplicity comes at a cost in terms of accuracy for two reasons; first of all, since each input token is considered independently of others, the context in which the token appears is not taken into account, leading the algorithm to perform poorly with homographs. Secondly, if a token was not seen during the training stage, then it will not be present in the mapping from token to POS tag. The tagger must implement some form of fallback to handle such situations, such as labelling the token as having the POS tag ‘unknown’ or choosing a tag based on morphological features of the word.

2.3.2 Hidden Markov Model

A Markov Model is a probabilistic model of a system represented as a finite state automaton. When combined with a set of observations about a real world instance of the modelled system, it can be used as a Hidden Markov Model (HMM), where the current state of the real world system can be predicted by a sequence of observations that have been recorded and fed into the model.

In order to train a HMM tagger, statistics are gathered on preceding and following tags in the training corpus, and are used to construct a graph, where nodes represent POS tags, and edges represent the probability of transitioning between tags. Weights are then assigned to the edges of the graph using the forward-backward algorithm.

In order to tag a set of input tokens we compute the probability of each token being assigned each possible

---

Table: Tag Frequencies

<table>
<thead>
<tr>
<th>Token</th>
<th>Tag Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'AJ0': 6, 'NN1': 1, 'AJ0-NN1': 3</td>
</tr>
<tr>
<td>Leading</td>
<td>'VVG': 29, 'AJ0-VVG': 11, 'AJ0': 25</td>
</tr>
<tr>
<td>The</td>
<td>'AT0': 3434</td>
</tr>
<tr>
<td>Whole</td>
<td>'AJ0': 97, 'AJ0-NN1': 18, 'NN1': 23</td>
</tr>
</tbody>
</table>

Figure 2.3: A subset of a BNC-derived mapping from input tokens to the frequency of assigned POS tags. The rightmost column contains counts of how many times the corresponding token was assigned each POS tag in the training corpus. Though only the most common tag is needed for the frequency based model described in Section 2.3.1, the frequency counts for other tags are often retained since they can be used by later stages of the TM pipeline.

---

14 These mappings are usually large; for the whole BNC corpus, the mapping is around 10MB.
15 Each word can be stored in a hash table where put and get operations have a time complexity of \(O(1)\).
16 Usually given the code ‘UNK’ or ‘UNC’.
17 For example, if a word is capitalised and not at the start of a sentence, then it is likely a proper noun.
18 A probabilistic model is one where each possible state in the output state space is assigned a probability as to how likely it is to occur. For POS tagging applications, each input token is given a probability of being assigned to each POS tag.
19 A Finite State Automaton (FSM) is a directed graph where the vertices of the graph represent the set of states that a system can be in, and the edges represent transitions between states.
20 For example, the probability that an adjective follows another adjective (‘big blue’) might be higher than that of a determiner following an adjective (‘horrible the’).
21 The forward-backward algorithm is a dynamic programming algorithm that efficiently computes the edge weights in a HMM graph in two passes (one forward pass, one backwards pass).
CHAPTER 2. PART OF SPEECH TAGGING

POS tag (based on a mapping such as the one in Table 2.3), and then use the Viterbi algorithm\(^\text{22}\) to find the most likely route\(^\text{23}\) through the graph from the start state to the end state [4].

More advanced HMM taggers look at bigrams or trigrams\(^\text{24}\) which helps them to take the context of a word into account, and thus become more accurate by helping them differentiate between different homonyms [5].

### 2.3.3 Rule Based Model

A rule based tagger seeks to formulate a set of rules that dictate which POS tags to assign to tokens. These rules are then applied one after another to tokenized text, such that tags are iteratively assigned to tokens and the assignment is refined on each application of a rule. Approaches differ in the kinds of rules they use; some rules apply to text that has no structure other than having been tokenized, but most approaches aim to improve the accuracy of the tags that have already been assigned to text.

Rule based taggers are largely concerned with generating rules, as opposed to the trivial task of applying them; since there are a large number of possible rules, selecting the best subset to apply is a computationally complex task. The most popular rule based tagger is the Brill tagger [6], which is described fully in Section 2.5.

### 2.4 Choosing a Model

Only one POS tagging model was implemented and parallelised as part of the project due to the complexity of the algorithms involved (and thus the implementation time required). As such, the choice of which algorithm to implement had the capacity to greatly affect the outcome of the project. Table 2.1 critically compares the different algorithms according to four criteria, and from this, the selection was reduced to just two choices; a Hidden Markov Model tagger and a Brill tagger.

Since the aim of the project was not to implement a state of the art tagger, but rather to parallelise a widely used tagger, both are valid choices. Of the two, the Brill tagger was chosen for three reasons; firstly, it is conceptually easy to understand which makes it easier to reason about when parallelising. For example, it is easy to test that a set of rules are being applied as they should be. Secondly, the algorithm has a significant computational complexity that will provide a basis for performance improvement through parallelisation. Finally since a Brill tagger depends on another ‘base tagger’, other applications could conceivably use the parallelised Brill tagger on top of an existing tagger as an extra stage in their TM pipelines in order to further improve tagging accuracy.

### 2.5 Brill Tagging

Previously, Brill’s Transformation Based Learning (TBL) algorithm was introduced and chosen as the POS tagging method to be implemented and parallelised. This section explains Brill’s TBL algorithm, as well as detailing the motivations behind it.

\(^{22}\) The Viterbi algorithm finds the most likely sequence of states visited in a path through a Markov Model given a sequence of observed events.

\(^{23}\) Hence the most probable sequence of tags.

\(^{24}\) A bigram or trigram are groups of two or three data items respectively that are examined at the same time in order to gain more contextual information about the data. HMM’s that use bigrams or trigrams are called second and third order models respectively.
Table 2.1: Comparison of POS Tagging Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Easily parallelised</th>
<th>Currently used</th>
<th>Computationally complex</th>
<th>Usable with existing TM pipeline</th>
<th>Probabilistic output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency Based Tagger</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hidden Markov Model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Support Vector Machine</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cyclic Dependency Network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Neural Network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Maximum Entropy Model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Brill Tagger</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Comparing the seven different POS tagging algorithms examined according to five traits; how easily they are parallelised, if they are currently used (and have not become obsolete), if they are computationally complex enough to warrant work on performance optimisation through parallelisation, if they easy to integrate into an existing Text Mining pipeline, and if the output is probabilistic (required in a minority of TM pipelines).

Brill’s work can be split cleanly into two parts; training a model and applying the model. Here, both are explained individually; first the application of the model is discussed, and then the training of the model is outlined. This order is chosen so that the reader has a clear understanding of the desired result before having to consider how training works.

2.5.1 Algorithm Overview

Brill’s algorithm aims to improve the output of an existing part of speech tagger (thereafter called the base tagger) by applying a set of rules specific to the base tagger’s output that will fix common errors that it makes. The key insight that Brill’s algorithm gives, is how to find a good set of rules to apply to the output from the base tagger.

2.5.2 Applying the Model

In the case of Brill’s TBL algorithm, the model consists of a set of rules that are sequentially applied to tagged input data in order to increase the accuracy of the tagging. For every rule in the model, the algorithm iterates over the input data, and for each token, replaces the token’s assigned tag with the rule’s tag if all of the rule’s conditions are satisfied.

To implement this functionality, rules consist of two parts; a set of conditions which dictate when the rule should be applied, and a tag that will replace a token’s existing tag when the conditions of the rule are satisfied. An example rule is shown in Figure 2.4.

25Such as always tagging a specific word wrong.
CHAPTER 2. PART OF SPEECH TAGGING

[WORD[-1] == "the",
WORD[0] == "start"] -> "NOUN"

Figure 2.4: An example Brill rule with two prerequisite conditions. This rule is activated if the previous word is equal to “the” and the current word is equal to “start”. If these conditions are met, then the current word (“start”) is given the tag “NOUN”. This reflects that “start” is sometimes a noun (e.g. “The start of the race”), but is sometimes another part of speech (e.g. “start the car”).

A model produced by the Brill training algorithm consists of many rules, which are applied sequentially over the corpus. Sequential application is important, since rules often correct errors introduced by preceding rules, an example of which is shown in Figure 2.5.

[TAG[0] == "UNK"] -> "NOUN"
[TAG[-1] == "NOUN",
TAG[0] == "NOUN",
TAG[1] == "DETERMINER"] -> "VERB"

Figure 2.5: Two Brill rules where the second rule fixes errors introduced by the first rule. The first rule encapsulates the fact that most words are nouns (around 40%), so if the base tagger was unable to tag a word, then tagging as a noun is most probably correct. The second rule recognises that if the previous word is a noun, and the following word is a determiner (e.g. “the”) the current word could have been unknown before the previous rule changed it to a noun, yet it is more likely to be a verb such as “Jim read the book”.

Brill’s algorithm typically increases the accuracy of existing tagging applications by around 5 – 10% [7].

2.5.3 Training the model

In order to apply the Brill tagging algorithm, a set of rules needs to be constructed that are specific to the characteristic errors produced by the base tagger. The set of possible rules is large; any word or part of speech tag can be used in the prerequisite conditions, and each rule can have up to three conditions. Consequently, finding the best rules by simply enumerating them is not feasible. The key insight of Brill’s algorithm, is that it generates rules based on errors made by the base tagger rather than enumerating all possible rules. This is done by iterating through a base-tagged corpus, and instantiating rules to fix each error using a set of template rules, as shown in Algorithm 1. Once the rules have been generated, a score is assigned to each based on how effective it is at improving the assigned tags in a test corpus, and the best rule is added to the model, as illustrated in Algorithm 2. This process is repeated until enough rules are generated, or time constraints mean that the training be stopped.

26The number of possible rules is exponential in the number of words in the target language, which is usually very large.
Algorithm 1 Pseudocode for generating Brill rules from an existing corpus that has been tagged by a base tagger. Line 2 initialises a set to store candidate rules that are then generated by iterating over the corpus and instantiating template rules with contextual information of each error found (lines 3 – 9).

Algorithm 2 Pseudocode for Brill’s training algorithm. Lines 2 and 3 initialise variables to keep track of the best rule out of all the candidate rules. The for loop on line 4 iterates through each rule and assigns it a score. This is done by incrementing the score variable when the rule corrects a mistake (lines 10 – 11), and decrementing the variable when the rule ‘breaks’ a tag (lines 12 – 13). The algorithm returns the rule with the best score that it found (lines 17 – 22).
2.6 Chapter Conclusion

This chapter examined seven different POS tagging algorithms and evaluated their applicability to this project against five criteria. Brill’s transformation based learning algorithm was chosen as the most suitable implementation candidate because of its widespread use, amenability to parallelisation and portability to other text mining pipelines through its reliance on a base tagger.
Chapter 3

Parallelisation

3.1 Chapter Overview

Previous sections have focused on the text mining techniques being implemented, but not on the parallelisation aspect of the project. This section aims to provide the context in which to frame the implementation details later on in the document, explaining the motivations behind parallelisation and the methods through which it can be achieved.

3.2 Background

Since the 60’s, the speed of computer processors has doubled every eighteen months in line with Moore’s law[27]. For software engineers, programmers and computer users, this means that the effective speed of running computer programs increases over time as new hardware becomes available. It is however, theoretically impossible for the clock speed of processors to carry on increasing in such a manner indefinitely for the following reasons [9]:

- Since the technology used to fabricate[28] transistors onto silicon chips is progressing, the feature size[29] is decreasing. This means that the transistor count of chips is increasing, thus giving CPU designers more flexibility and capacity to implement more logic and faster chips. However, the point where one transistor consists of just a few atoms and therefore cannot be further reduced in size is approaching, and meaning that the trend of increasing transistor counts while maintaining a constant die area cannot continue.

- As the size of each transistor decreases, the amount of transistors per unit area in a CPU increases proportionally. Since each transistor generates heat as it changes state, the heat generated per unit area increases at the same rate, which becomes prohibitive[30].

---

27 ‘Moore’s law’ refers to an observation and prediction made by Gordon Moore in 1965 [8], where he stated that the number of transistors in integrated circuits was doubling every 18 months.

28 Fabrication is the process used to create integrated circuits.

29 The smallest size that a transistor can be fabricated onto a chip.

30 If the processor becomes too hot, then it can become irreversibly ‘heat damaged’. As a consequence, many CPU’s now feature ‘turbo mode’, where they will increase their clock speed for a short duration before backing off to avoid overheating. Other techniques to manage heat production include turning sections of the processor off (clock gating) and better cooling...
Unable to gain further increases in the clock speed of their CPU designs, chip manufacturers have turned to other techniques in order to continue the exponential speed increases in processors. One such method is to make chips have multiple cores; if a new processor runs at the same clock speed as its predecessor but has two cores, then it can, in theory, process twice as many instructions as the previous processor did.

Unfortunately, this is not a perfect solution; the performance of any single program is not increased since the clock speed remains constant, and the program’s thread of execution can only be executed on a single core at any one time. In order to make programs faster, software engineers can no longer wait for new hardware releases to do the work for them; programs have to be parallelised to run on multiple cores concurrently in order to take advantage of the new hardware. In fact, parallelism is being labelled as the new Object Oriented Programming by some, since it is seen to be the next paradigm shift in how programs are developed.

While multithreading may seem like a good solution upon first inspection, it is rare in practice that a speed up proportional to the number of processing cores can be obtained. This is due to the fact that not all portions of a program can be parallelised, meaning that some aspects of the execution must remain sequential. This was formalised by Amdahl [11], and later revised by Gustafson [12], to give an equation by which the theoretical performance increase from parallelisation can be calculated:

\[
\text{Performance increase} = 1 - p + sp
\]  

Where:
- \( p \)  The percentage of the program that can be parallelised.
- \( s \)  The speed up obtained in the parallelised parts of the program.

During the course of this project, three different levels of parallelism have been exploited by the author, and are described in the following subsections.

### 3.3 Parallelisation on a Single Thread

Though clock speeds cannot increase much further, chip designers have implemented limited forms of parallelisation inside the processor cores themselves. These methods aim to parallelise the instructions issued to the processor while preserving the linear semantics of the computation. Technologies implementing these ideas include SIMD instructions, out-of-order execution and hyperthreading. All of these methods of parallelism are performed automatically by the system, and are consequently of less relevance to the project, though a brief overview is given in Appendix A.2.

---

31 In this manner, the performance of the processor can be doubled in line with Moore’s law.

32 Object Oriented Programming is a software engineering paradigm introduced in the early 1990’s, and was a major shift for many programmers in terms of how they wrote software.

33 Inherently sequential operations such as IO, or non-associative mathematical operations cannot be parallelised very easily.

34 Single Instruction Multiple Data [13].
3.4 Parallelisation with CPU Threads

Unfortunately, the conservative nature and limited scope of the automatically applied, single threaded parallelisation techniques discussed in the previous section limits their effectiveness in practice. Linear performance gains relative to the available parallelism are rarely achieved.

The most common technique for implementing parallelisation on one machine is multi-threading. If an application has multiple threads, then it can take advantage of the multiple execution cores that may be available on a machine. Two techniques for running multiple threads of execution inside one process were examined; POSIX threads are covered in Appendix A.3 and the parallel patterns library is covered below.

3.4.1 Parallel Patterns Library

As with most hard problems in software engineering, the implementation details common to most multi-threaded programs have been largely abstracted away by various libraries. Examples include the java.util.concurrent library, and multiprocessing for Python.

As will be explained and justified in Chapter 4, C++ was chosen as the implementation language for the project, and so the Parallel Patterns Library (PPL) was used to implement multithreading. PPL was developed by Microsoft so that developers can transition from single threaded C++ code to a multithreaded implementation very easily.

PPL exploits the natural structure inherent in programs to make it easy for the developer to indicate that certain statements should be parallelised. An example of a code block that is trivially parallelised is that of Figure 3.1, which is a prime candidate for multithreading, since each iteration of the for loop is independent of the others. This can be trivially parallelised by PPL, as shown in Figure 3.2. The library determines a threading strategy at runtime, and distributes the work evenly over all the available cores of a CPU.

```c
int[] importantArray = new int[50];
for (int i = 0; i < 50; i++) {
    importantArray[i] = i % 2;
}
```

Figure 3.1: Initialising an array of fifty integers in a serial manner. In this example, array cells with odd indices are given the value 1, and cells with even indices are given the value 0.

---

35 Other techniques are available, many of which involve spawning multiple processes rather than using threads.
36 This includes the logical CPU cores presented to the operating system with hyperthreading technology.
37 This is advantageous since the runtime environment varies between different machines and depending the other computations running on the host machine at the same time. Other libraries targeted at more specific implementations such as machines with dedicated compute hardware such as Intel’s Xeon Phi may be less general in their approach.
 CHAPTER 3. PARALLELISATION

```c
#include <ppl.h>

int[50] importantArray = int[50];

concurrency.parallel_for (0, 50, [&importantArray](int i) {
  (*importantArray)[i] = i % 2;
});
```

Figure 3.2: Initialising an array using the Parallel Patterns Library (PPL), where the specific details of the threading (such as specifying how many threads to create) are abstracted away from the programmer. The operation is parallelised by way of a function call to the `parallel_for` function (line 4), where the loop parameters are specified. Note that in this specific case, it may be more efficient to do the work in a serial manner for two reasons; the computation being done is minimal and setting up threads has some overhead which could outweigh the actual cost of computation, and running multiple threads on the same data may cause high rates of cache invalidation on multiple processor cores [14].

Though it is possible to write multithreaded code with a judicious use of synchronization logic and very simple datastructures like the array in Figure 3.2, most datastructures found in the C++ standard library like hash tables, trees or lists are not designed for multi-threaded access. Using these constructs with PPL’s `parallel_for` loops and `parallel_for_each` loops, would most likely result in many issues relating to data corruption. Consequently, PPL has thread safe implementations of the most commonly used datastructures in the C++ standard library, including concurrent vectors, concurrent maps and concurrent queues.

### 3.5 GPU Accelerated Computation

Since the early 2000’s many computers have shipped with Graphics Processing Units (GPU’s); chips specialised to handle the massive amount of calculations required to render 3D graphics. As a result of being heavily optimised for graphics applications, the architecture of a GPU is fundamentally different from that of a CPU, and programmers have in the past, had to go to great lengths in order to take advantage of this extra computational capacity [15]. In recent architectures, GPU’s have included various technologies enabling them to be used as commodity massively parallel co-processors for consumer applications. The next sections will introduce how GPU’s are optimised for their intended workload, discuss the technologies available to programmers in order to leverage the power and parallelism inherent in GPU’s, and look at what problems GPU’s are best suited to solving.

#### 3.5.1 Design Goals of the GPU

Before the architecture of the GPU is examined, we must first understand the design goals it aims to satisfy so that the architectural decisions make sense. The three main requirements for a GPU are [17]:

---

38 Due to multiple threads trying to read and write to multiple locations in memory at the same time, or internal values being updated concurrently causing illegal states to occur.

39 ‘Thread Safe’ is the term commonly used to describe datastructures, objects and other programming constructs that are designed to operate normally when used concurrently by multiple threads.

40 Since general purpose GPU-computation libraries did not exist until 2007, problems had to be mapped onto a sequence of graphics operations that could be performed on the GPU.

41 A coprocessor is an additional processor used in conjunction with the main processor (usually the CPU), often specialised towards a specific task such as graphics processing (GPU’s) or IO management [16].
---
• Deliver large amounts of compute. This is required to render the millions of pixels present modern high-definition displays.

• Provide substantial parallelism. The operations required of graphics hardware are well suited to parallelism due to the large-array based nature of the computations, often requiring the same transformation to be applied to every element.

• Maximise throughput over latency. While one frame every 16ms is sufficient for humans to experience smooth video, computer processors operate in the order of nanoseconds. This disparity between the granularity of the human perception and the speed of processors means that the latency of any given graphics operation is of little importance.

As a result, GPU’s have evolved to have a large amount of total compute, with the latest cards having capacity to perform over 4Tflops as well as having a memory bandwidth of 288GB/sec. This dwarfs the compute available on similar CPU chips.

3.5.2 Architecture of the GPU

Critical requirements of most CPU designs include to make a single stream of instructions run as fast as possible, and to have a low latency on any single instruction. However, as presented above, the goal of a GPU is to maximise throughput at the possible expense of latency. As a result of this, accesses to memory can have a very high latency so long as the GPU advances as many threads as possible on every clock cycle.

This is done by maximising the number of cores on the GPU so that many parallel streams of execution can be used to hide the memory latency, ensuring that there are always eligible threads waiting to be run, and by implementing zero-overhead thread switching in the GPU hardware.

As a consequence of this, highly parallelizable tasks are well suited to the GPU since they can be run over many threads and have a high total throughput. In the context of this project, rules are derived from a large corpus of one hundred million words; if the corpus is split into blocks, and a thread processing one block of the words is stalled, then other threads that are waiting to run can immediately do so instead. This has the

42 4K Video has a resolution of 3840x2160, and if the system aims to provide video output at 60 frames per second, then the GPU must process around 500 million pixels per second. Each pixel will take many operations to produce, meaning the number of operations performed will be orders of magnitude larger.

43 60Hz frame rate is sufficient to saturate the human visual system [18].

44 The Tesla K40 is described as having 4.29Tflops (1Tflop is equal to $1 \times 10^{12}$ floating point operations per second) of single precision floating point performance on NVIDIA’s website [http://www.nvidia.com/object/tesla-workstations.html], last accessed 06/03/2016).

45 The Tesla K40 is described as having a memory bandwidth of 288GB/sec (with error correction turned off) on NVIDIA’s website [http://www.nvidia.com/object/tesla-workstations.html], last accessed 06/03/2016).

46 Intel’s E7-8890 v3 processor is listed as having a memory bandwidth of 102GB/sec [http://ark.intel.com/products/84685/Intel-Xeon-Processor-E7-8890-v3-48M-Cache-2_50-GHz], last accessed 06/03/2016) and is reported to perform at 1.5Tflops [http://www.nextplatform.com/2015/05/05/intel-puts-more-compute-behind-xeon-e7-big-memory/], last accessed 06/03/2015).

47 Modern GPU’s have dedicated on-board random access memory, but sometimes memory is shared with the host computer’s main memory which is far slower. In both cases, access is expensive compared to processing speed and can take many clock cycles.

48 The time taken to execute any given thread is not a priority.

49 When a thread stalls (due to a long operation such as memory access), or the operating system decides to run another thread on the processor, its state must be saved before another thread can be (re)initialised and started on the same processor core (also known as a context switch). On a CPU, this is a costly operation that involves actions such as copying the values of registers to memory, and properly managing the state of IO operations being executed by the thread. Since thread switching is a common operation on GPU’s, zero overhead thread switching is implemented, which means that it only takes one clock cycle to switch which thread is executing on a core.
effect of increasing the overall throughput of the program, since the GPU cores spend the majority of their time processing, not context switching or waiting on stalled threads.

There are many more optimisations implemented by GPU designs that were studied and exploited in the project. This included using read only ‘constant’ memory, coalescing memory accesses with GPU-optimised data structures, easing register pressure and maximising occupancy.

The results of the contrasting design goals between CPU’s and GPU’s are clearly visible in Figure 3.3 where the die area allocated to different components in the processors is visibly different.

![Figure 3.3](image)

**Figure 3.3:** Since the GPU is designed to tolerate high memory latency, it has a far smaller die-area allocated to on-board caches (shown in orange). Since the primary use case for GPU’s is mathematical operations such as vector processing for graphics, the control logic (yellow) is far smaller than in the CPU, since these operations do not require advanced features such as branch prediction and instruction pre-fetching (explained in Appendix A.5). The GPU also features vastly many more ALU’s (shown in green) than the CPU, since its ALU’s are far less complicated and are controlled in blocks of 32 (also requiring less control logic). [40]

### 3.5.3 CUDA

NVIDIA CUDA is a programming platform created in order to standardise GPGPU computation, allowing programs running on the CPU to utilise the compute capacity of one or more massively data-parallel GPU coprocessors. CUDA processors make use of the SPMD model of computation, where all threads have the same set of instructions and access the same address space, but may take different paths of execution depending on their assigned data elements [19].

Though there are libraries that will compile code from other programming languages into a CUDA-compatible form,

---

50 Architecture-aware datastructures and algorithms exhibit memory access patterns that involve using neighbouring memory locations at the same time. Since the width of the bus between the GPU processor and its memory is often that of multiple machine words, several adjacent memory reads or writes can be serviced by one access.

51 Each Streaming Multiprocessor (SM) has a finite number of registers, therefore reducing how many registers are used by each thread maximises the number of threads that can run on a single SM.

52 Occupancy is a measure of GPU processor utilisation, calculated by dividing the activity of each processor by its maximum achievable activity.

53 Compute Unified Device Architecture.

54 General Purpose GPU (computation).

55 Single Program Multiple Data; an extension of the Single Instruction Multiple Data (SIMD) paradigm. The same program is run on multiple different instances of the input data concurrently.
The CUDA framework is an extension of the C programming language, and while libraries exist to automatically convert code from other programming languages into CUDA enabled C [20], most software projects use a language with native support. The programmer uses the CUDA syntax to markup functions, named ‘kernels’, that are to be run in a massively parallel manner on the GPU. Each kernel is started on many threads simultaneously, with each thread able to access its own unique ‘thread id’ to determine what data in the global address space it should operate on. By enabling the programmer to specify which functions should be run on the GPU, CUDA isolates the data-parallel portions of the program from the more sequential areas, which often results in a more modular, and loosely coupled codebase.

Since a kernel can be launched on literally billions of threads, CUDA provides a way to manage the complexity of coordinating them all. A block is a group of up to 1024 threads that are able to synchronize their execution using primitives such as atomic operations and barriers. In addition to defining how many threads should constitute a single block, the programmer must also decide on a grid size, meaning that the number of threads launched on the GPU is implicitly defined.

Choosing the parts of a program that are to be parallelised on the GPU is an important design decision; only a small subset of programs are suited to efficient parallelisation on such a platform. The following considerations must be taken into account when partitioning the program [19]:

- The executing kernel should be able to take advantage of the zero-overhead thread switching implemented on GPU’s. Enough threads should be launched so that there are always eligible threads waiting to be launched since threads can often be stalled due to the high memory latency.
- Using specialised on-chip memory such as read-only constant memory can enhance performance, since less communication is required with global memory.
- Threads should be grouped so that they run similar execution paths. In order to simplify control logic, instructions are issued to warps and if the flow control of some threads in a warp diverges, then other threads in the block will have to execute NOP instructions until flow merges again and the same instruction stream can be issued to all threads.
- Inter-thread synchronisation should be minimised. Hardware limitations mean that threads can only be synchronized within a block, which lets the hardware be scalable, but limits any parallelism involving complex inter-thread communication.
- The GPU maintains a separate address space from that of the CPU, and as a consequence, the programmer must copy the data from the CPU address space into the GPU address space. This is

---

56 Only C, C++ and Fortran have native CUDA support at this time.
57 A barrier is a synchronisation primitive that requires all threads to have reached a specific point in their execution before they can continue.
58 Blocks are defined in a coordinate based system in either one, two or three dimensions. This makes certain problems easier to reason about, for example, computing physics in 3D might be easier with a 3d grid of blocks. In the Appendix, Figure C.2 and Figure C.3 show different configurations of blocks sizes and dimensions.
59 The total number of threads launched is equal to the number of threads per block multiplied by the number of blocks.
60 Only programs with large amounts of data parallelism, little inter-thread synchronisation, and long sequences of simple operations are ideal for GPGPU computation, though programs with only some of these attributes can still obtain performance benefits.
61 GPU’s have a five tier memory hierarchy, with registers being fastest, and global memory being slowest. Constant memory is a fixed-size, read-only memory that must be initialised before a kernel is started, but is situated closer to the GPU chip than the main memory is and is heavily cached.
62 A thread warp is a group of 32 threads that have the same instructions issued to all of them. Grouping threads into warps simplifies the logic required inside the GPU since it means that thread scheduling can be done at a less granular level (on the level of warps instead of on the level of individual threads).
63 No-operation instructions are instructions that occupy one clock cycle of a processor, but do no useful work. An example of this is the ARM instruction MOV R1 R1, which moves the value of register 1 into register 1.
most efficiently implemented with a single call to move a large chunk of memory (such as an array), rather than moving many smaller chunks of memory (such as individual structs or bytes).

3.6 Chapter Conclusion

This chapter investigated different methods of parallelisation, examining the PPL library for CPU based parallelism, and the CUDA framework for GPGPU computation. In addition, a high-level description of the current state of GPU architecture was provided, which was necessary in order to introduce the theory behind the analysis and optimisations discussed in the following chapter.
Chapter 4

Experiments and Analysis

4.1 Chapter Overview

This chapter presents the experiments conducted during the course of the project, describing their motivation, design decisions, implementation and results. A total of three experiments were run, as shown in Table 4.1. Information on the machine used to run the experiments is given in Appendix A.4.

Table 4.1: Tabulation of Experiments

<table>
<thead>
<tr>
<th>Experiment name</th>
<th>Fastest method</th>
<th>Performance increase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collatz</td>
<td>GPU</td>
<td>13.4×</td>
<td>Parallelisation of a basic computational task.</td>
</tr>
<tr>
<td>Brill tagging #1</td>
<td>Parallel CPU</td>
<td>3.5×</td>
<td>Parallelisation of a real-world text mining algorithm; Brill rule training.</td>
</tr>
<tr>
<td>Brill tagging #2</td>
<td>GPU</td>
<td>100.8×</td>
<td>The second iteration of parallelising the Brill training algorithm.</td>
</tr>
</tbody>
</table>

Comparison of the three main experiments carried out in the project; their outcomes and short descriptions.

4.2 Experimental Design and Methodology

Before each of the experiments is introduced, explained and analysed, an overview of the methodologies used when carrying out the experiments is given, along with a discussion of the technology choices and design decisions made in the project.

4.2.1 Programming Language

A key decision in any software project is that of what programming language to use. Requirement FR7 (“Parallelise POS tagging-training on the GPU”) somewhat simplifies the decision, since many languages do
CHAPTER 4. EXPERIMENTS AND ANALYSIS

Table 4.2: Programming Language Comparison

<table>
<thead>
<tr>
<th>Name</th>
<th>GPU Bindings</th>
<th>Conceptual level</th>
<th>Prior experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3rd party library</td>
<td>High</td>
<td>10K LOC</td>
</tr>
<tr>
<td>Java</td>
<td>3rd party library</td>
<td>High</td>
<td>70K LOC</td>
</tr>
<tr>
<td>C</td>
<td>Native with CUDA</td>
<td>Low</td>
<td>10K LOC</td>
</tr>
<tr>
<td>C++</td>
<td>Native with CUDA</td>
<td>Low (with high level features)</td>
<td>None</td>
</tr>
<tr>
<td>Fortran</td>
<td>Native with CUDA</td>
<td>Low</td>
<td>None</td>
</tr>
<tr>
<td>Scala</td>
<td>3rd party library</td>
<td>High (functional)</td>
<td>2K LOC</td>
</tr>
</tbody>
</table>

C++ was chosen as the programming language to use for the project since along with C and Fortran, it is most directly supported by the CUDA framework (other languages usually rely on runtime compilation to C code that is then ran on the GPU), and C++ is low level (giving lots of control over the computation which is important for performance tuning), while still retaining high level constructs such as objects and templates (both C and Fortran lack this). The only disadvantage to C++ is that the author had no prior experience with it. However, this was seen as an opportunity to learn a new programming language, and was not seen as a great risk, since the author has experience picking up new programming languages, especially since C++ is similar in a paradigmatic sense to other languages already known by the author.

not have the facility to run arbitrary code on the GPU. Table 4.2 shows a comparison of the candidate languages, and describes the decision to use C++.

Since the project involved lots of text manipulation, programming languages and tools most suited to dealing with large amounts of text data were used in addition to writing programs in C++. In particular, Python and Bash were used to analyse the BNC corpus and Java was used to transform the corpus from one format to another in Experiment #3.

4.2.2 Development Tools

Source control was considered a top priority for the project, due to the fact that it helps manage the complexity of implementing multiple features concurrently, and it is able to let the developer rewind the source to any point in the history. Git was used as the source control system for the project because of its wide adoption in the open source world, and its inherently distributed model that is well suited to development on multiple systems.

The Visual Studio 2015 IDE was used to develop the implementation due to it having features such as being able to easily use C++ libraries such as boost and being integrated with NVIDIA’s NSight CUDA debugger. For the minority of work that was done in Java or on the command line, either the Eclipse IDE was used (which has excellent integration with the unit testing framework JUnit) or emacs was used to edit Python and Bash scripts.

---

64 Though many programming languages provide ways to manipulate graphics, general purpose GPU computation is not a core feature of most programming languages.
65 In order to determine optimum parameters for Experiment #3.
66 This suits the author’s workflow, since development was done in university, on two separate laptops and on two separate test machines. With Git, getting the project’s source code onto a new development machine is as simple as cloning the repository onto the machine.
67 Integrated Development Environment.
68 Used for unit testing and for its additional datastructures.
69 Conventional debugging tools such as gdb do not work well for CUDA programs since they cannot monitor execution on the GPU. The NSight debugger lets developers step through code as it is running and profile a GPU kernel to identify bottlenecks.
70 https://www.gnu.org/software/emacs/
4.2.3 Testing

Testing is fundamental to software projects, and a variety of approaches were used for this codebase, which included unit testing of modules, and end-to-end testing\textsuperscript{71} of whole sub-systems. Tests were formulated to target a variety of possible bugs, and exercise as many different execution paths of the code as was possible. The ‘boost’ testing framework employed in the C++ codebase had integrated memory leak detection, which was particularly useful when working with large datasets, since even small leaks can become an issue in a long running program.

Of course, since the overarching aim of the project was to parallelise, and therefore speed up text mining algorithms, performance testing was of critical importance to the project. As such, a number of different benchmarking methods were employed, such as the unix ‘\texttt{time}’ utility and timing libraries in C++. Experiments were run multiple times, and the results from each averaged to ensure that random error in observed performance results did not adversely affect the analysis.

4.3 Experiment 1: Parallelising Computation of the Collatz Conjecture

This experiment aimed to investigate the feasibility of parallel CPU and GPU computation when applied to a simple, easy to implement task; verifying the collatz conjecture. Although the outcomes from this experiment served as a guide for designing the following experiments, the detail for this experiment is included in Appendix A.6 due to it not being directly relevant to the parallelising text mining algorithms.

4.4 Experiment 2: Parallelising the Scoring of Brill Rules

With Brill’s method of POS tagging, training the model is far more expensive than using it in practice due to having to first generate rules\textsuperscript{72} before then assigning a score to each based on how it performs over the whole test corpus. Since large numbers of rules can be generated, scoring them all becomes prohibitive in terms of computation time.

Taking into account the lessons learned from the Collatz experiment, the aim of this experiment was to investigate to what extent training a Brill tagger could be parallelised to give a performance boost. Again, the experiment explored both CPU based and GPU based parallelism in order to speed up the computation, and compared their performance to that of a serial implementation.

4.4.1 Experiment 2 - Implementation

Brill training is a compute intensive task, with a runtime of $O(n \times e \times m)$ where $n$ is the size of the training corpus, $m$ is the number of rule templates and $e$ is the number of errors in the corpus\textsuperscript{73}. If the training

\textsuperscript{71}These ‘integration’ tests were particularly important, since they ensured that different parts of the program (that had been tested in isolation by unit tests) worked properly together.

\textsuperscript{72}Generating rules based off errors in a corpus has a complexity that is linear in the size of the input corpus.

\textsuperscript{73}It should be noted that this is the runtime for training one rule, and that a model will typically consist of around 50 to 500 rules in practice, though this is largely irrelevant in the current analysis since the scoring process is just repeated until enough rules are generated.
corpus is small and accurately tagged\textsuperscript{74}, then training will be quick\textsuperscript{75}, however, a small training corpus is undesirable since it may cause overfitting\textsuperscript{76} of the to occur. The aim of this experiment is to increase the performance of Brill training so that larger training corpora can be used, and thus more general rules be generated.

In order to decide how best to parallelise the Brill training algorithm, each step of the algorithm was analysed:

- **Generate rules based off the errors in the tagged training corpus:**
  Generating the rule set has a complexity of $O(em)$ where $e$ is the number of errors in the training data, and $m$ is the number of rule templates. In the worst case, every token in the training data is incorrectly tagged, so $e = n$. Since $m$ is small\textsuperscript{77}, the computational complexity of this step is largely linear in the number of errors in the training data.

- **Score each generated rule:**
  For each rule to be given a score, it must be applied to every token in the training data, and its score adjusted as to whether it corrected the tag, did nothing, or assigned an incorrect tag from a previously correct tag. In the worst case, $em$ rules could be generated, each of which would be scored against $n$ training tokens, giving a runtime of $O(n \times e \times m)$. Due to each rule being scored against the whole corpus, this is far the most compute intensive step.

An implementation of the Brill training algorithm was written in C++, and parallelised with the PPL library by scoring each rule in parallel\textsuperscript{78}. This produced similar performance increase to the Collatz experiment in Appendix A.6.3. This was ported to use a GPGPU model of computation using the CUDA framework, which involved copying the data to the GPU prior to starting the computation and using simple arrays\textsuperscript{79} rather than more complex data types and data structures such as strings and hash tables. The score for each rule is written to a large array indexed by the rule number, which was copied back to main memory for use by the rest of the training algorithm. It should be noted that a key trade-off in the design of GPGPU software is that of striking the balance between fast computation on the GPU device, and slow data transfers to and from the device memory.

### 4.4.2 Experiment 2 - The Proposed Hypothesis

The expected result from this experiment was that the large amount of parallelism accessible to the GPU and the inherent data-parallelism present in the Brill training algorithm\textsuperscript{80} would mean the GPU implementation would outperform either the serial or parallel CPU implementations. This hypothesis assumes that the lower clock speed of the GPU would be offset by the greater parallelism it provides, yet since data is required to be

---

\textsuperscript{74}Tagging here refers to the tags assigned by the base tagger; a more accurate base tagger requires fewer corrections from the Brill tagger.

\textsuperscript{75}Few mistakes in the training corpus means that there will be fewer rules generated. A smaller training corpus means that scoring each rule takes less time since there are fewer words to evaluate against.

\textsuperscript{76}‘Overfitting’ is the term used to describe when a Machine Learning model reflects the random error in its training dataset, rather than encapsulating a generalised representation of the data. In this case, if the training corpus was small, an incorrectly tagged word (due to manual tagging error) could cause an erroneous rule to be generated.

\textsuperscript{77}Typically, between 10 and 50 rule templates are used for Brill tagging.

\textsuperscript{78}As described above, only rule scoring need be parallelised since rule generation takes a negligible amount of time in comparison.

\textsuperscript{79}Though there are libraries that support advanced data structures such as hash tables for GPGPU computation, the CUDA framework supports only primitive arrays by default.

\textsuperscript{80}Each rule can be scored independently of the rest, meaning that parallelism can be applied with little or no performance penalty.
transferred to and from the GPU from main memory before computation on the device can occur, a slight constant overhead associated with data transfer was predicted for the GPU operations.

### 4.4.3 Experiment 2 - Results, Analysis and Interpretation

Unfortunately, the results obtained from the experiment did not support the hypothesis, since the GPU implementation was found to exhibit performance far worse than the single-threaded serial implementation running on a CPU, as shown in Figure 4.1. This was a distressing result, since considerable research effort had been spent learning CUDA and implementing a tagger using it, only for it to be outperformed by a ‘normal’ CPU implementation.

However, a deeper analysis of the problem revealed aspects of the Brill training algorithm that were not optimal for GPGPU execution:

- The algorithm processes the input data at the sentence level, and the sentences in the training data can be of variable lengths. This is non-optimal for implementation on the GPU, since as described in Section 3.5, each parallel thread is running on a separate CUDA core, which are all executing in lockstep. If some cores finish processing a sentence faster than others, they sit idle until the other cores complete their thread.

- Words and sentences are also of variable length, forcing threads to deviate from the same execution path, which causes some threads to be idle while others process longer sentences.

- Since CUDA is a subset of C, it does not include all of the standard libraries that a C/C++ programmer would normally have access to. String processing functions such as strlen and strcmp were among those left out, and so had to be written by hand, making them likely to exhibit suboptimal performance when compared with the heavily optimised versions in the standard library used by the CPU implementation.

Once the analysis of the algorithm had been completed, the exact changes required in order to better optimise the algorithm needed to be identified.

The NVIDIA NSight debugger was used to profile the rule scoring kernel in order to be able to further optimise the next prototype. Figure 4.2 shows the number of warps able to be scheduled per multiprocessor on the GPU with varying block sizes. Figure 4.3 shows how often the scheduler had multiple warps ready to run, and Figure 4.4 shows how many instructions were executed per clock cycle.

---

81 An operation is defined as applying one rule to one token of data. The number of operations per iteration of Brill’s training algorithm is equal to the number of rules being scored multiplied by the size of the corpus.

82 Some sentences in the BNC have over 200 words (these are often quotes or other non-standard text).

83 NSight is part of NVIDIA’s CUDA drivers and is integrated into Microsoft Visual Studio.

84 As described in section 3.5.3, execution on the GPU is split into a grid, and each block in the grid is split into threads. All threads in a block are scheduled to run in lock-step on the same multiprocessor.
CHAPTER 4. EXPERIMENTS AND ANALYSIS

Figure 4.2: The NVIDIA NSight debugger analysed the number of warps that could be run in parallel on each SM (Streaming Multiprocessor). This is affected amongst other things, by the number of threads in each warp (more threads means that more resources are required to manage the threads, and the SM only has a finite amount of resources). In the experiment, each warp was made up of 1024 threads, meaning that only 32 could run in parallel on a single SM (as shown in the diagram). If 192 threads per block were used instead, then 48 warps could have been executed concurrently on a single SM, which would be 50% more efficient.

Figure 4.3: The GPU has a scheduler that decides which warps should run on which processing cores at any given time. This graph shows the percentage of times when the scheduler had no warps to issue, or only one to issue. This indicates that the GPU spent over 55% of the time idle.

Figure 4.4: A key metric for the efficiency of any processor is how many instructions are executed per clock cycle. An issued instruction is one that is started by the processor, and an executed instruction is one that is completed successfully. Due to each thread in a warp executing in lock step, and not all threads being at the same point in the execution path, over 50% of issued instructions are aborted in order to ensure the processors do not deviate from executing in lock-step, making the effective clock speed of the GPU only 160MHz in this instance.
4.4.4 Experiment 2 - Concluding Remarks

Though the experiment didn’t provide the results that were expected, much was learned from analysing the performance and behaviour of the running program. It was possible to link the runtime statistics from the NSight debugger to the architectural quirks of the GPU as discussed in Section 3.5.

4.5 Experiment 3: Reformulating the Brill Rule Scoring Problem

Although the results from the first experiment were disappointing, there was enough development time to spare for a re-implementation of the tagger. This time, a more architecture-aware approach could be used, integrating the lessons learnt from analysis of the previous attempt.

The goals of this experiment were similar to those of experiment #2, however now the aim was specifically to boost the performance metrics provided by the NSight debugger, in particular:

- Getting the IPC value close to 1.0.
- Reducing the number of times the scheduler could not schedule warps.
- Finding an optimum number of threads per block to initialise the kernel with.

Of course, the overall goal is to produce a rule scoring system that can outperform a serial CPU implementation.

4.5.1 Experiment 3 - Implementation

The performance of the previous GPU tagger was limited by the three factors identified in Section 4.4.3, so it was obvious to make changes intended to fix these issues:

- The varied sentence lengths in the corpus caused each thread to take a different execution path which is non-optimal for CUDA programs. The solution is to make all the sentences the same length in the training data by pre-processing the corpus to select only sentences of a certain length.

- Each word having a different length also caused threads in the same warp to take different execution paths. This was fixed by mapping each word in the training data onto a unique integer in the same pre-processing step as when the sentence pruning was done, which also has the effect of compressing the size of the corpus, making data-transfer time between the GPU and CPU cheaper. This was inspired by how some programming languages implement an optimisation called ‘string interning’ in order to reduce the memory usage of immutable character arrays.

- The problem of having to implement string operations in CUDA code was mitigated as a side effect of mapping each word to its own integer, since now only integers need to be compared.

Of course, if all words were mapped onto integers for the GPU, comparing the existing CPU implementations was no longer fair, so they were converted to use integers instead of words too.

---

85 This is a fast linear-time operation performed with a Java program.
86 Note that this does not affect the quality of the generated rules, because the rule scoring step can be applied multiple times to corpuses of different sentence lengths and the scores added together.
87 This is also a linear-time operation, since adding an element to a map and looking up elements in the map can be implemented in $O(1)$ time with a hash table.
Additionally, memory-hierarchy optimisations were employed in order to further enhance the performance of this second prototype. In particular, rule templates were identified as being frequently accessed in a read-only manner by many threads in order to instantiate new rules when an error was identified in training data. These rules were consequently placed in constant\textsuperscript{88} GPU memory to reduce access time.

4.5.2 Experiment 3 - The Proposed Hypothesis

It was expected that an efficiency boost would be observed for the second Brill tagging attempt, since each GPU core will be processing data with exactly the same dimensions and therefore cores will not need to wait if they finish processing early. This should let the IPC value increase since each core will spend less time waiting for the others, and should let the scheduler issue more warps since existing warps will complete faster.

Processing integers instead of strings should give a performance increase in any case, since there is less data to be handled\textsuperscript{89}.

4.5.3 Experiment 3 - Results, Analysis and Interpretation

The performance of the new implementation of rule scoring on the GPU was much more successful, as shown in Figure 4.5. Since the bottlenecks inherent in the previous GPU implementation were removed, in particular the need for cores to wait while other cores pursued a divergent path of execution was removed, so that the GPU could take full advantage of the parallelism in its cores.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure4_5.png}
\caption{The performance of the second GPU implementation of the rule scoring algorithm is 12x faster than the serial CPU implementation, and 110x faster than the original GPU implementation, performing just under 100,000 scoring operations per second.}
\end{figure}

Despite the second prototype performing well, it was as-yet unknown whether it was possible to obtain more performance from the GPU. In order to determine this, the NSight debugger was used to detail the fine-grained runtime characteristics of the CUDA kernel, and compare it with the previous GPU implementation.

The simplest optimisation performed in the second prototype was to choose an optimal number of threads per block. The NSight debugger is able to simulate the effect of varying block sizes and determine which block sizes are optimal for the running kernel. As shown in Figure 4.2, decreasing the block size from 1024 to 192 increased the number of warps able to be scheduled on each SM\textsuperscript{90}.

\textsuperscript{88}Constant memory (described in more detail in Section 3.5), is a small amount of immutable memory with fast access time, originally designed for storing textures in GPU image processing.

\textsuperscript{89}Each multi-character word is now mapped to a single integer. If the average word length is 8 (which is actually 9 characters since character arrays are null terminated) then it will take 72 bits of memory whereas single integer will take up just 32 bits; a 55% decrease.

\textsuperscript{90}Each SM is scheduled a pool of many warps, and a single warp is scheduled for execution whenever the SM has enough free resources to run it and it is ready for execution (i.e. no thread in the warp is stalled). Since any stalled thread inside a warp can prevent the whole warp from being executed, it is advantageous to have many warps assigned to a SM so that there is a high likelihood that one warp will execute, and the SM will not be idle.
CHAPTER 4. EXPERIMENTS AND ANALYSIS

Figure 4.6: The warp issue efficiency in the second prototype (Figure 4.6b) is far greater than that of the first prototype (Figure 4.6a). In the first prototype, no warp could be scheduled 55% of the time, whereas in the second prototype this was only the case 4.5% of the time. In the second prototype, there were at least two warps ready to be scheduled nearly 80% of the time, and one warp ready just over 15% of the time. Since only one warp is required for execution to happen, this means that the GPU was able to perform useful computation in 95% of cases, as opposed to just 45% in the first prototype.

Since each SM had more warps ready to be scheduled at runtime, the situation where there were no warps were ready arose less frequently, meaning that less time was spent idle. This is shown in Figure 4.6 which compares the warp issue efficiency in the first prototype with that of the second prototype.

The efficiency of a computation is closely tied to the number of instructions executed per clock cycle (IPC), so this was an important metric to optimise for in the second Brill implementation. Figure 4.7 shows a comparison between the IPC achieved in the first prototype against the IPC achieved with the second prototype. The increase in IPC means that the computation is around 600% more efficient\(^1\) with less clock cycles being wasted waiting on lock-step related execution dependencies.

It should be noted that the number of issued instructions that failed to complete successfully was 55% in the first GPU implementation, but just 3% in the second one. This suggests that a more architecture-aware implementation enabled the scheduler to better predict and mitigate execution dependencies\(^2\).

\(\text{\(91\) Computational efficiency is important in two ways here, in terms of time and energy. While the purpose of this project is to reduce the time it takes to train a Brill Tagger, energy use is important as well since if these techniques were deployed in either a datacenter running thousands of instances of a program or on a mobile device such as a laptop or smartphone, then energy consumed per operation is important.}\)

\(\text{\(92\) These can include instruction fetching, memory dependencies, execution dependencies, waiting for synchronisation between cores, memory throttling etc.}\)

4.5.4 Experiment 3 - Concluding Remarks

This experiment was far more successful than the first GPU tagging experiment, not only surpassing the serial CPU tagging performance, but also fully utilising the resources available on the GPU\(^3\). As a result, this prototype was deemed sufficient to be used as the GPU accelerated tagger in the final analysis of parallelisation techniques.

4.6 Conclusions of Experiments

This section summarises the different implementations and compares their performance, before analysing the suitability of each for deployment in real-world text mining pipelines. Suitability analysis is mainly concerned with examining the flexibility and effectiveness of an implementation.

---

\(\text{\(91\) As empirically observed using profiling statistics from the NSight debugger.}\)
Figure 4.7: The IPC values for the second prototype are far higher than those for the first, with the executed IPC increasing by 596%, which translates directly to faster execution of the program. Furthermore, the number of aborted instructions is far lower in the second prototype (3% for the second prototype, but 56% for the first prototype), suggesting that the threads executing in lockstep on different processors rarely diverged into different execution paths. Note that the percentage of aborted instructions can be found by using the formula: Aborted = 100 − (\frac{100}{\text{Issued}} \times \text{Executed}).

The results in Figure 4.8 show the relative performance difference between the Brill rule scoring implementations produced in this project, where the trend is that as the parallelism increases, the performance of the system increases too. Though CPU parallelisation is by far the most commonly exploited method of concurrent execution of a single program on current PC’s, the near four-fold performance increase obtained using this technique was dwarfed by the twelve and hundred-fold performance boosts observed with GPU co-processing.

Though this suggests that GPU processing may be the ‘next step’ for making programs run faster, it should be noted that porting the algorithm to run on the GPU was the product of over two months of development work, while parallelising the algorithm on the CPU is far simpler and takes just a few hours.

Furthermore, it was only after thorough research and many attempts that the original Brill tagging algorithm was mapped onto a form that was computable in an efficient manner on the GPU. This is a good example of the inflexibility of GPGPU computation in its current form; not only does the problem have to be suited in the sense that it should have lots of data-parallelism, but it should also work with the limitations of the CUDA platform.
Figure 4.8: The relative performance difference between the serial, parallel, and GPU implementations of Brill rule scoring. The two rightmost values are the same implementation running on different hardware; GPU96 was ran on an NVIDIA Quadro 600 96 core device, and GPU512 was run with an NVIDIA GeForce 580 device, which has 512 CUDA cores and a 772MHz clock speed.
Chapter 5

Conclusion

This chapter summarises the report, aiming to discuss the achievements of the project, along with what could be done in the future in order to expand the work.

5.1 Discussion of Achievements

The aim of this project was to examine how parallelisation can be used to increase the performance of training a model for Brill’s part of speech tagging algorithm. This required extensive research into a variety of topics including linguistics (both historical and current POS tagging algorithms), parallel processing techniques using a variety of libraries, and GPU Architecture.

This research was carried out and implemented using a range of different technologies; C++ for implementing Brill’s training algorithm, Java for corpus pre-processing, as well as Bash and Python for statistical analysis of corpus content. Prior to undertaking the project, the author hadn’t had exposure to most technologies used in the project. This necessitated learning C++, as well as the CUDA and PPL libraries that were used to parallelise the algorithms in order to obtain performance increases over running the code sequentially.

The results obtained (up to 100x performance increases in scoring Brill rules) strongly support the experiment hypothesis, but this was only the case after extensive profiling, prototyping, and empirical testing was applied in order to map the problem into a form amenable to GPGPU computation.

5.2 Further work

Thus far, this project has successfully parallelised a single stage in what would be a multi-stage text mining pipeline. Though some stages are trivial enough not to warrant parallelisation, later stages of the pipeline could be parallelised in order to speed up the overall process further. Examples of this could be generating parse trees from tokenized and tagged text, detecting which input tokens correspond to entities in some knowledge database, or extracting relations between tokens in the text.

If the later stages of the TM pipeline were implemented using a ML approach, then both training and testing could be parallelised as necessary too. In some cases, either stage may be cheap and therefore not warrant
parallelisation, such as the application of Brill rules in this project.

Nevertheless, techniques used by this project such as string interning could be applied to other parts of a TM pipeline in order to map the computation onto the GPU. Programs created in this project to map input strings onto integers and back again could be used directly in further work with little or no modification.

Acknowledgements

The author would like to thank Dr. Richard Neville for his guidance during the project, Dr. Andy Carpenter for his supervision as second marker and Professor Allan Ramsey for his help with understanding natural language processing concepts. The author would also like to acknowledge the use of the BNC corpus, the size of which enabled this project to push the available GPGPU hardware to its limits.
Bibliography


Appendix A

Background Information

In this appendix chapter, additional background information is included in order to give additional context to the rest of the document if it is required.

A.1 Additional POS Tagging Algorithms

A.1.1 Cyclic Dependency Network

Cyclic Dependency Network taggers use a variety of different signals to determine the tag for each word, and in particular, consider both the preceding and following contexts of the word [21]. This works well, since most other tagging algorithms work from left to right, and only determine a tag based on tokens that have already been seen (the left context), thus ignoring the potentially useful information that could be found by examining the tokens after the current token. Just like other tagging algorithms, lexical features and statistical prior knowledge derived from corpus analysis are used to assist tagging.

A.1.2 Multilayer Perceptron

Neural networks are versatile machine learning algorithms that can be applied to a vast array of problems, since they can approximate any continuous function\(^95\), a property that has led them to be called \textit{universal approximators} [22]. This makes them useful for finding good solutions to NP-complete\(^66\) and NP-hard\(^97\) problems\(^98\). Because the computational complexity of the POS tagging problem cannot be bounded by a polynomial [26], neural networks are a good choice of learning algorithm to apply, since they can approximate a good solution within a reasonable time.

\(^{94}\)Lexical features include capital letters, morphemes, suffixes, affixes etc.

\(^{95}\)A function over \(I \rightarrow \mathbb{R}\), where \(I = \{a, b\} = \{x \in \mathbb{R} \mid a \leq x \leq b\}\).

\(^{96}\)NP is the class of problems where a solution can be verified in polynomial time on a deterministic Turing Machine [23]. An NP-Complete problem is a problem that is both in NP itself, and is at least as hard as any other problem in NP, since there must be a polynomial time reduction from any problem in NP to the NP-Complete problem.

\(^{97}\)An NP-Hard problem is at least as hard as every other problem in NP, such that there is a polynomial time reduction from any problem in NP to the NP-Hard problem, but the NP-Hard problem does not have to be in NP itself.

\(^{98}\)For example, neural networks have been used to solve classic arcade games [24], which have been proven to be NP-hard [25].
In particular, neural networks are good for learning applications where there is not enough training data available [27]; this is not usually an issue when training a tagger to work with English, since there are many large corpora available. However, other languages (such as Portuguese, used in the paper) have no such corpora, and training data must be created manually, which imposes a limit on the amount available at a reasonable cost.

### A.1.3 Support Vector Machine

Support Vector Machines (SVM’s) have been a popular choice for the POS tagging application due to their linear time complexity when training, and the flexible way that they can use features extracted from words [28]. Since SVM taggers typically trade off accuracy for efficiency, they were not found to be suitable for this project as there is little need for parallelisation of algorithms that require relatively little computation.

### A.1.4 Maximum Entropy Model

A Maximum Entropy Model is a generalisation of logistic regression that is not restricted to binomial classification. Maximum entropy models were popular for part of speech tagging in the mid to late 1990s, but their accuracy has since been surpassed by that of more recent machine learning algorithms [37, 38].

### A.2 Parallelisation on a Single Thread

Many processors implement some forms of parallelism even when only a single thread is running. These techniques are often automatically performed in a manner that is completely transparent to the programmer and user.

#### A.2.1 SIMD Instructions

SIMD instructions are specialised instructions that can perform the same operation on multiple data items. For example, the ARM instruction ADD8 can compute four 8-bit additions in one CPU cycle, since the machine has a 32-bit words and hence the additions can be done in parallel. Compilers are often able to recognise situations in which a SIMD instruction is more appropriate than multiple SISD instructions, and this optimisation is almost always applied with no input from the programmer.

---

99 SVM taggers can have a flexible number of input features, which makes them versatile taggers since the features examined can be easily changed for different use cases. Examples of this might be having different features for different languages (e.g. Arabic might have a different feature set to English).

100 A method of classifying input data into two classes.

101 I.e. it can sort input into more than two classes.

102 A word is the size of data operated on by a processor’s instruction set. The architecture of a computer system is often optimised with the size of one word in mind, for example, memory fetches will return values of a width equal to a multiple of the word size to optimise cache utilisation.

103 Single Instruction Single Data.
A.2.2 Out of Order Execution

Out-of-order execution is where the processor will execute instructions ahead of the current instruction in the execution thread if the current instruction is blocked due to a hazard and the pending instructions must be independent of the current instruction. Instructions are considered independent if neither refers to data accessed by the other instruction [29].

A.2.3 Hyperthreading

Hyperthreading lets an operating system take advantage of a superscalar architecture inside a CPU. Each processor core presents itself as having two logical cores, and the operating system assigns instruction streams to both, meaning that two threads are scheduled at once on the same physical core.

Unlike having two physical cores, the logical cores share most resources (the execution engine, caches, system bus etc), though some duplication may be present (such as the ALU). This is a double edged sword, since if one thread stalls, the other can use all of the core’s resources for that period. However when both threads are active, then resources must be shared, causing both threads to compete for them.

A.3 POSIX Threads

POSIX Threads (often referred to as Pthreads), are the ‘standard’ way to have more than one thread of execution inside one process, and are most commonly used in low level languages like C. The Pthread specification is defined in the IEEE Standard 1003, and is widely implemented across many operating systems, including Linux, Mac OS X and Microsoft Windows.

The Pthread API is very powerful, but remains lightweight in its resource usage [30]. It contains operations that implement all the functionality of threads (creating, destroying etc) as well as operations to control their flow (mutex locks, condition variables and barriers).

The power and fine grained control that the Pthread API gives programmers is offset by the mental overhead required of them. When implementing a multi-threaded program, the programmer must be careful to thoroughly reason about any synchronisation logic in order to avoid problems such as deadlock, since though deadlock can be detected at runtime, there are no known techniques for resolving the situation [31].

As a consequence of the above issues, programming with Pthreads imposes a significant mental overhead upon the programmer. One technique to mitigate this is to exploit the natural data parallelism inherent in some problems in such a way that communication between threads is minimised. For example, finding the sum of all elements in an array requires almost no inter-thread interaction; given \( n \) threads, the array can be split into \( n \) equal parts and each thread can find the sum of its corresponding part. Once the threads have

---

104 A hazard is some runtime constraint that prevents a processor from re-ordering the instruction stream to speed up program execution. An example of this is that an add operation cannot be done before the load operation that will load the relevant data into registers.

105 A superscalar architecture is one that allows multiple instructions to be executed in the same clock cycle using techniques such as pipelining.

106 A Mutex (mutually exclusive) lock is a construct that prevents two entities (in our case, threads) from accessing the same resource at the same time. Only one entity can hold a lock to the resource at once.

107 A thread can wait at a condition variable until the variable is set (by another thread) in such a way that some condition is met, in which case the waiting thread can continue.

108 Data parallelism involves applying the same computation to many different data items.
completed, the final answer can be obtained by summing their results. Problems such as this, where there are few dependencies between sub-problems are most amenable to threaded computation.

However, problems dependent on the use of exclusive resources (such as some IO operations, locks etc.) are hard to solve efficiently with threads, since the contested resource often becomes a bottleneck [32], meaning that the computation must be carefully managed to avoid this.

### A.4 Machine Specifications for Running the Experiments

Unless otherwise stated, the machine being used to run the experiments has an Intel® Core™ i5-4460 (quad-core) CPU running at 3.20GHz, 8GB of RAM and a NVIDIA Quadro 600 GPU, clocked at 640MHz and consisting of 96 CUDA™ cores. It should be noted that the relative calibre of the CPU and GPU are vastly different; the i5 CPU was released in 2014 priced at $187.00, while the GPU was released in 2010, costing $179.00 (now worth, $40). Despite being priced similarly, a comparable graphics card from 2014 has roughly four times the performance.

### A.5 Execution Optimisations on the CPU

This section provides a brief overview of CPU architecture features to aid the performance of single-threaded programs.

#### A.5.1 Branch Prediction

Branch prediction a technique used to speed up program execution on CPU’s by predicting the outcome of a branch instruction before it is executed. This means that instructions at earlier stages of the CPU pipeline can be executed as normal until the outcome of the branch instruction is known (in which case the predicted instructions will have to be thrown away if the predictor made a mistake).

#### A.5.2 Instruction Prefetching

CPU processors often request instructions from main memory before they are actually needed so that they will be present in the cache when they are due to be executed (thus avoiding the thread having to stall while the instructions are fetched from memory).
A.6 The Collatz Conjecture Experiment

A.6.1 Background - The Collatz Conjecture

The Collatz conjecture is a mathematical observation that if you continually apply the Half or Triple Plus One rule to an integer, then it will eventually reach 1 as shown in Figure A.1. The rule is a function that is repeatedly applied to an integer such that it eventually reduces the integer to 1. The function is defined as:

\[ f(n) = \begin{cases} 
\frac{n}{2}, & \text{if } n \mod 2 = 0 \\
3n + 1, & \text{otherwise}
\end{cases} \]  

(A.1)

Informally, the rule divides the number by two if it is even, and if it is odd, then the number is tripled and incremented.

We can use a computer to automatically apply the rule to an input number and see how many applications of the rule (referred to as iterations) are required before the number reaches 1. The time taken to compute the iteration count\footnote{How many iterations are required to reach one.} of the Collatz conjecture for all integers below some limit \( n \) can be measured, and can serve as both as a good vehicle for learning the parallelisation technologies to be used in the project, and also as a proof of concept benchmark to compare the performance of different methods of parallelisation. The reasons behind choosing to implement the Collatz conjecture as a test method are:

- The work corresponding to each integer from 1 to \( n \) can be computed in parallel with no data dependencies\footnote{It should be noted that partial computations could be cached (i.e. memoization could be implemented) in order to speed up the computation. Implementations listed in this section do not use memoization to optimise the code, though this could be implemented in the future.}.

- The conjecture is easy to implement, meaning that it can be applied to a wide range of functions.
APPENDIX A. BACKGROUND INFORMATION

The parallelism was implemented gradually [33], first writing the serial implementation, then parallelising it using multiple CPU threads with PPL, and finally porting the code over to CUDA to be run in a massively parallel manner on the GPU. Algorithmics for the CPU implementations (serial and parallel) are shown in Algorithm 3 and those for the GPU implementation are shown in Algorithm 4.

Algorithm 3

The algorithm for computing the collatz conjecture on the CPU for all numbers between 0 and max. The main for loop at line 3 iterates through every number between 0 and max, and computes the collatz conjecture for that number. The while loop at line 6 continues until the \( v = 1 \) (or an overflow occurred, meaning \( v \) would be negative), and applies the collatz transformation as defined by the conjecture at lines 8 and 10. The number of times the while loop has iterated is stored in \( i \), and when the number does reach 1, this is written into an array at the index specified by the original number (line 14). This array is returned at the end of the procedure (line 16).

Algorithm 4

The algorithm for computing the collatz conjecture on the GPU. The range of numbers computed is equal to the number of threads launched, since each thread computes the number equal to its thread id. Other than not iterating through each number in the range and the fact that no value is returned (rather the result is stored in the result array, \( x \)), this version of the algorithm is equal to the CPU version in Algorithm 3.

A.6.2 Collatz Experiment - The Proposed Hypothesis

It is expected that the CPU will achieve near optimal single-threaded performance since the computation does not require many reads and writes from memory\(^{114}\) and though there are some conditional instructions\(^{115}\), CPU architecture is well optimised for this eventuality [34].

Parallelising the algorithm to map the computation onto all available cores was trivial using the PPL library. Since the function used to compute the stopping time of the Collatz conjecture is side-effect-free\(^{116}\), it can be trivially parallelised by using a PPL parallel for loop to call the Collatz computing function for every desired integer. Since there is no interdependency between the instructions, the author expected a linear

---

\(^{114}\)Since there are only two variables used in the algorithm, the state of the whole computation should be able to be kept entirely in registers, which is very fast.

\(^{115}\)For example, the CPU doesn’t know whether to divide by two or multiply by three and add one until it has tested for evenness, and therefore the pipeline may stall.

\(^{116}\)Calling `collatz(i)` computes and returns the iteration count for \( i \), and does not alter any other state in the program, meaning that multiple threads can call the function concurrently and not interfere with each other.
Figure A.2: The relative computation time for computing the Collatz conjecture on the CPU on a single thread, a CPU with multiple (4) cores, and on a GPU with 96 cores. Note that the speedup observed by the GPU is not of the order of 96 times since each core is significantly less powerful than those found on the CPU, and time is taken transferring the data between main memory and the GPU device memory.

The performance difference between the CPU serial and parallel implementations are to be expected; distributing the work over four cores is likely to give a performance boost of around four times, especially since this problem qualifies as embarrassingly parallel.\(^\text{119}\)

Likewise, a computational task requiring little or no inter-thread communication, as well as being composed of mostly simple operations is ideal for the GPU. The only characteristic of the Collatz conjecture that makes it less suitable for GPGPU computation is that the sequence of operations is different for each input number, meaning cores with data not satisfying a conditional instruction must be turned off for a clock cycle, potentially doubling the runtime of the program.

A.6.3 Collatz Experiment - Results, Analysis and Interpretation

The algorithm was run with \(n = 10^6\), on the GPU, CPU with a single thread and CPU with multiple threads. The total time taken to perform the computation was recorded, including setting up the result arrays and copying data between the main memory and the GPU memory. The computation was done ten times, and the average runtime recorded for each level of parallelisation. The relative speed differences for this experiment are presented in Figure A.2.

The performance difference between the CPU serial and parallel implementations are to be expected; distributing the work over four cores is likely to give a performance boost of around four times, especially since this problem qualifies as embarrassingly parallel.\(^\text{119}\)

---

\(^\text{117}\)Divisions, or multiplications and additions.

\(^\text{118}\)Assuming an intelligent and accurate compilation, the runtime would double in the worst case since there are two possible operations for each iteration of the Collatz function.

\(^\text{119}\)An embarrassingly parallel problem is defined as one that can “easily be divided into components that can be executed concurrently” [35].
One might be surprised at the GPU performance in Figure A.2; despite the GPU having 96 cores, the performance increase is only around 13 times. This is likely due to the following factors:

- There is an overhead in GPU computation associated with transferring data to and from the device [36]. For this specific problem, the overhead is low since the data being transferred is an integer array and thus can be completed in one DMA-accelerated operation.

- Each CUDA core on the GPU is vastly less powerful than a CPU core. As stated in the introduction to Section 4, the CUDA cores are clocked at 640MHz, five times slower than a CPU core. Furthermore, as stated in Section 3.5.3, the control logic in each GPU core is very simple, and doesn’t implement the performance optimisations present in CPU cores.

A.6.4 Collatz Experiment - Concluding Remarks

In this section a performance evaluation of different methods of parallelisation for computing the Collatz conjecture were presented; single threaded code, multi-threaded code using PPL and GPU-accelerated code using CUDA, of which the latter performed best. The general principles for analysing the runtime of GPGPU programs were shown, and hypotheses were constructed to explain the obtained results.

---

120 Direct Memory Access is a technique used to optimise data transfer between different memory spaces in a computer system by offloading the transfer to a specialised processor separate from the CPU. This means that the CPU does not need to waste cycles on relatively trivial tasks such as copying memory (and the CPU caches are not trashed).
# Appendix B

## C5 tag set

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJ0</td>
<td>Adjective (unmarked) (e.g. GOOD, OLD)</td>
<td></td>
</tr>
<tr>
<td>AJC</td>
<td>Comparative adjective (e.g. BETTER, OLDER)</td>
<td></td>
</tr>
<tr>
<td>AJS</td>
<td>Superlative adjective (e.g. BEST, OLDEST)</td>
<td></td>
</tr>
<tr>
<td>AT0</td>
<td>Article (e.g. THE, A, AN)</td>
<td></td>
</tr>
<tr>
<td>AV0</td>
<td>Adverb (unmarked) (e.g. OFTEN, WELL, LONGER, FURTHEST)</td>
<td></td>
</tr>
<tr>
<td>AVP</td>
<td>Adverb particle (e.g. UP, OFF, OUT)</td>
<td></td>
</tr>
<tr>
<td>AVQ</td>
<td>Wh-adverb (e.g. WHEN, HOW, WHY)</td>
<td></td>
</tr>
<tr>
<td>CJC</td>
<td>Coordinating conjunction (e.g. AND, OR)</td>
<td></td>
</tr>
<tr>
<td>CJS</td>
<td>Subordinating conjunction (e.g. ALTHOUGH, WHEN)</td>
<td></td>
</tr>
<tr>
<td>CJT</td>
<td>The conjunction THAT</td>
<td></td>
</tr>
<tr>
<td>CRD</td>
<td>Cardinal numeral (e.g. 3, FIFTY-FIVE, 6609) (excl ONE)</td>
<td></td>
</tr>
<tr>
<td>DPS</td>
<td>Possessive determiner form (e.g. YOUR, THEIR)</td>
<td></td>
</tr>
<tr>
<td>DT0</td>
<td>General determiner (e.g. THESE, SOME)</td>
<td></td>
</tr>
<tr>
<td>DTQ</td>
<td>Wh-determiner (e.g. WHOSE, WHICH)</td>
<td></td>
</tr>
<tr>
<td>EX0</td>
<td>Existential THERE</td>
<td></td>
</tr>
<tr>
<td>ITJ</td>
<td>Interjection or other isolate (e.g. OH, YES, MHM)</td>
<td></td>
</tr>
<tr>
<td>NN0</td>
<td>Noun (neutral for number) (e.g. AIRCRAFT, DATA)</td>
<td></td>
</tr>
<tr>
<td>NN1</td>
<td>Singular noun (e.g. PENCIL, GOOSE)</td>
<td></td>
</tr>
<tr>
<td>NN2</td>
<td>Plural noun (e.g. PENCILS, GEESE)</td>
<td></td>
</tr>
<tr>
<td>NP0</td>
<td>Proper noun (e.g. LONDON, MICHAEL, MARS)</td>
<td></td>
</tr>
<tr>
<td>NUL</td>
<td>The null tag (for items not to be tagged)</td>
<td></td>
</tr>
<tr>
<td>ORD</td>
<td>Ordinal (e.g. SIXTH, 77TH, LAST)</td>
<td></td>
</tr>
<tr>
<td>PNI</td>
<td>Indefinite pronoun (e.g. NONE, EVERYTHING)</td>
<td></td>
</tr>
<tr>
<td>PNP</td>
<td>Personal pronoun (e.g. YOU, THEM, OURS)</td>
<td></td>
</tr>
<tr>
<td>PNQ</td>
<td>Wh-pronoun (e.g. WHO, WHOEVER)</td>
<td></td>
</tr>
<tr>
<td>PNX</td>
<td>Reflexive pronoun (e.g. ITSELF, OURSELVES)</td>
<td></td>
</tr>
<tr>
<td>POS</td>
<td>The possessive (or genitive morpheme) ’S or ’</td>
<td></td>
</tr>
<tr>
<td>PRF</td>
<td>The preposition OF</td>
<td></td>
</tr>
<tr>
<td>PRP</td>
<td>Preposition (except for OF) (e.g. FOR, ABOVE, TO)</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>PUL</td>
<td>Punctuation - left bracket (i.e. ( or [)</td>
<td></td>
</tr>
<tr>
<td>PUN</td>
<td>Punctuation - general mark (i.e. . , : ; ? ... )</td>
<td></td>
</tr>
<tr>
<td>PUQ</td>
<td>Punctuation - quotation mark (i.e. ‘ ’ ” )</td>
<td></td>
</tr>
<tr>
<td>PUR</td>
<td>Punctuation - right bracket (i.e. ) or ] )</td>
<td></td>
</tr>
<tr>
<td>TO0</td>
<td>Infinitive marker TO</td>
<td></td>
</tr>
<tr>
<td>UNC</td>
<td>&quot;Unclassified&quot; items which are not words of the English lexicon</td>
<td></td>
</tr>
<tr>
<td>VBB</td>
<td>The &quot;base forms&quot; of the verb &quot;BE&quot; (except the infinitive), i.e. AM, ARE</td>
<td></td>
</tr>
<tr>
<td>VBD</td>
<td>Past form of the verb &quot;BE&quot;, i.e. WAS, WERE</td>
<td></td>
</tr>
<tr>
<td>VBG</td>
<td>-ing form of the verb &quot;BE&quot;, i.e. BEING</td>
<td></td>
</tr>
<tr>
<td>VBI</td>
<td>Infinitive of the verb &quot;BE&quot;</td>
<td></td>
</tr>
<tr>
<td>VBN</td>
<td>Past participle of the verb &quot;BE&quot;, i.e. BEEN</td>
<td></td>
</tr>
<tr>
<td>VBZ</td>
<td>-s form of the verb &quot;BE&quot;, i.e. IS, 'S</td>
<td></td>
</tr>
<tr>
<td>VDB</td>
<td>Base form of the verb &quot;DO&quot; (except the infinitive), i.e.</td>
<td></td>
</tr>
<tr>
<td>VDD</td>
<td>Past form of the verb &quot;DO&quot;, i.e. DID</td>
<td></td>
</tr>
<tr>
<td>VDG</td>
<td>-ing form of the verb &quot;DO&quot;, i.e. DOING</td>
<td></td>
</tr>
<tr>
<td>VDI</td>
<td>Infinitive of the verb &quot;DO&quot;</td>
<td></td>
</tr>
<tr>
<td>VDN</td>
<td>Past participle of the verb &quot;DO&quot;, i.e. DONE</td>
<td></td>
</tr>
<tr>
<td>VDZ</td>
<td>-s form of the verb &quot;DO&quot;, i.e. DOES</td>
<td></td>
</tr>
<tr>
<td>VHB</td>
<td>Base form of the verb &quot;HAVE&quot; (except the infinitive), i.e. HAVE</td>
<td></td>
</tr>
<tr>
<td>VHD</td>
<td>Past tense form of the verb &quot;HAVE&quot;, i.e. HAD, 'D</td>
<td></td>
</tr>
<tr>
<td>VHG</td>
<td>-ing form of the verb &quot;HAVE&quot;, i.e. HAVING</td>
<td></td>
</tr>
<tr>
<td>VHI</td>
<td>Infinitive of the verb &quot;HAVE&quot;</td>
<td></td>
</tr>
<tr>
<td>VHN</td>
<td>Past participle of the verb &quot;HAVE&quot;, i.e. HAD</td>
<td></td>
</tr>
<tr>
<td>VHZ</td>
<td>-s form of the verb &quot;HAVE&quot;, i.e. HAS, 'S</td>
<td></td>
</tr>
<tr>
<td>VM0</td>
<td>Modal auxiliary verb (e.g. CAN, COULD, WILL, 'LL)</td>
<td></td>
</tr>
<tr>
<td>VVB</td>
<td>Base form of lexical verb (except the infinitive)(e.g. TAKE, LIVE)</td>
<td></td>
</tr>
<tr>
<td>VVD</td>
<td>Past tense form of lexical verb (e.g. TOOK, LIVED)</td>
<td></td>
</tr>
<tr>
<td>VVG</td>
<td>-ing form of lexical verb (e.g. TAKING, LIVING)</td>
<td></td>
</tr>
<tr>
<td>VVI</td>
<td>Infinitive of lexical verb</td>
<td></td>
</tr>
<tr>
<td>VVN</td>
<td>Past participle form of lex. verb (e.g. TAKEN, LIVED)</td>
<td></td>
</tr>
<tr>
<td>VVZ</td>
<td>-s form of lexical verb (e.g. TAKES, LIVES)</td>
<td></td>
</tr>
<tr>
<td>XX0</td>
<td>The negative NOT or N'T</td>
<td></td>
</tr>
<tr>
<td>ZZ0</td>
<td>Alphabetical symbol (e.g. A, B, c, d)</td>
<td></td>
</tr>
</tbody>
</table>

*Table B.1: The C5 tag set used in the BNC corpus [39].*
Appendix C

Diagrams

1D Grid of 1D Blocks

Figure C.1: A one-dimensional grid consisting of three one-dimensional blocks, each with four threads [41].
APPENDIX C. DIAGRAMS

1D Grid of 2D Blocks

Figure C.2: A one-dimensional grid of two-dimensional blocks, each consisting of twenty threads [41].

1D Grid of 3D Blocks

Figure C.3: A one-dimensional grid of three-dimensional blocks, each consisting of twelve threads. Note that both the grid and the blocks can be one, two or three dimensional [41].