ACCELERATING TEXT RELATEDNESS COMPUTATIONS ON GENERAL PURPOSE GRAPHICS PROCESSING UNITS

by

Duffy Angevine

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2015

This thesis is dedicated in loving memory to Beth Angevine.

Table of Contents

List of Tables

List of Figures

Abstract

This thesis investigates a novel approach for accelerating document similarity calculations using the Google Trigram Method (GTM). GTM can be performed as either a 1:1 comparison between a pair of documents, a 1:N comparison which occurs between one document and several others, or as an N:N comparison, where all documents within a set are compared against each other. Existing research in this domain has focused on accelerating the GTM on standard processors. In contrast, this thesis focuses on accelerating the performance of an N:N document relatedness calculation using a General Purpose Graphics Processing Unit (GPGPU).

Fundamental to our approach is the pre-computation of several static elements. These static elements are the GTM inputs: the documents to be compared, and the Google N-Grams. The Google N-Grams are processed to produce a word relatedness matrix, and the documents are tokenized. They are then saved to disk to allow for recall and are available for calculating document relatedness.

The mapping of the GTM to a GPGPU requires analysis to establish an effective system to transfer documents to the GPGPU, the data structures to be used in the GTM calculations, as well as an investigation into how to effectively implement GTM on the GPGPU's unique architecture.

Having designed a set of GPGPU methods we systematically evaluate their performance. In this thesis, the GPGPU methods are compared to a multi-core Central Processing Unit (CPU) method that acts as a baseline. In total, two different CPU methods and four different GPGPU methods are evaluated.

The CPU hardware platform is a workstation with a pair of 8 core Intel Xeon processors, retailing for approximately $10,000. The GPGPU platform is a Nvidia GeForce 660 GTX, worth approximately $200 at the time of purchase. We observe across a wide range of data sets that the GPGPU achieved between 40% and 80% of the performance observed on the multi-core workstation, at one fiftieth of the cost.

List of Abbreviations Used

ACM	Association for Computing Machinery
API	Application Program Interfaces
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPS	Documents Per Second
FLOPS	Floating Point Operations Per Second
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GTM	Google Trigram Method
GLSL	OpenGL Shading Language
HLSL	High Level Shader Language
LCS	Longest Common Substring
NLP	Natural Language Processing
PCIe	Peripheral Component Interconnect Express
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
STL	Standard Template Library
STS	Semantic Text Similarity
WPS	Words Per Second
WRM	Word Relatedness Matrix

Acknowledgements

The author would like to acknowledge the assistance of a few key individuals who kept the thesis on track:

Dr. Andrew Rau-Chaplin, for providing expertise and guidance without which, this thesis would have languished.

Jessica, for her tireless support and encouragement through out the process of developing and writing this thesis.

Chapter 1

Introduction

Text similarity, also known as document relatedness, is an important concept of the Natural Language Processing (NLP) field. It has many applications, such as information retrieval [5, 16], text categorization [5, 16], and document classification [5, 16, 18, 9, 6]. With an ever increasing number of available documents, the need to quickly and accurately perform document relatedness is essential.

This thesis investigates several methods to accelerate an unsupervised text similarity approach called the Google Trigram Method (GTM) [9]. The GTM was selected because it performs on par with supervised text relatedness approaches [9] without requiring the same degree of overhead as those supervised methods. The GTM can be applied in any language provided there exists a collection of N-grams for that specific language [9]. When applying the GTM approach to a large collection of documents, also known as a corpus, the calculation of text similarity requires significantly more resources. With the GTM, the computation of relatedness between two (2) documents is defined in terms of the pair-wise relatedness between the words in the documents.

Efficient algorithmic techniques and high performance computing (HPC) are required to reduce any observed run-time. This leads to a $\mathbf{O}\left(\mathbf{M}^2\right)$ time computing the relatedness between a pair of documents, with the assumption that word relatedness can be computed in $\mathbf{O}(1)$ time. To compute the relatedness between $\mathbf{N}$ documents, $\mathbf{O}\left(\mathbf{N}^2 \times \mathbf{M}^2\right)$ time is required. This means that the expected run-time is tied to not only the length of documents in the collection but also the number of documents in the corpus.

The GTM can be applied to a pair of documents (1:1 comparison), to a whole corpus (N:N comparison), or to subsets of a corpus (1:N comparison). While this thesis primarily focuses on calculations of N:N document relatedness, this approach to improve the performance of the GTM's computation can be applied to 1:1 and 1:N calculations as well.

Multi-core hardware is a system with a number of processors, or cores, that are able to partition an assigned workload across the total number of cores present. Traditionally, multi-core systems are thought of as of Central Processing Units (CPUs), however General Purpose Graphics Processing Units (GPGPUs) are in this domain as well. The GPGPU is a component of every modern desktop or workstation as a dedicated processor to render images on the display. Recent advances in architectural design and Application Program Interfaces (APIs) have allowed GPGPUs to undertake more than the task of rendering pixels, and have made them effective multi-core systems in their own right. Research has shown that applying various traditional computing problems to the GPGPU has demonstrated improvement [12] over CPU-based multi-core versions. The GPGPU's architecture is an attractive option for accelerating the GTM for the following reasons:

1. The financial cost of adding a GPGPU to an existing workstation is often significantly less than adding an additional CPU.

2. The performance of GPGPU architecture continuously increases the floating point operations per second (FLOPS) at a faster rate than new CPU architecture [10].

To accelerate the GTM, this thesis leverages a mix of algorithmic engineering techniques and design optimizations that take advantage of the GPGPU's architectural features. We present methods that exploit pre-computational processes and data structures residing in memory to effectively compute word relatedness between 1:1, 1:N and N:N documents. This thesis also explores the trade-offs of mapping the GTM to the GPGPU architecture via two distinct approaches to calculating the document relatedness, the Pair-Wise and One-to-N approaches, as well as how to effectively provide these approaches with the required inputs to perform the GTM. These inputs can be broadly grouped into two areas of focus: the data structures that are used to represent word-pair relatedness required by the GTM, and transference of the corpus to the GPGPU.

In addition to evaluating the data structures and transfer methods used by the GPGPU implementations of the GTM, this thesis evaluates the optimal GPGPU approach against a multi-core CPU approach similar to the one presented in [15].

Evaluation of the proposed implementations on both GPGPU and CPU is performed using two different corpora. The two corpora were selected to establish the universality of each approach, as one corpus is composed of relatively small documents and the other is composed of variable length (longer) documents. The corpora are used in a series of experiments, where various subsets of each corpus is provided to each implementation, and the observed time taken to produce the document relatedness for the given subset is recorded.

The results of these experiments reveal that the GPGPU approach, when run on a Nvidia GeForce 660 GTX, yielded a performance equivalent to 40% to 60% of the multi-core CPU implementation when run on a Linux server containing 256 GB main memory and 2 Intel Xeon ES-2650 processors, each with 8 cores and Hyper Threading enabled.

1.1 Thesis Document Structure

The remainder of this thesis is organized as follows:

1. Chapter 2 provides an overview of existing research to situate the theories and approaches fundamentally important to an understanding of this work, including an overview of the GTM method for text relatedness, and the architecture of the GPGPUs used.

2. Chapter 3 presents the design choices that influenced the approaches used to perform the GTM in this thesis, as well as the framework responsible for preparing the inputs for the GTM algorithm.

3. Chapter 4 provides specifics on how the GTM algorithm was mapped to the GPGPU, and discusses the analysis and design decisions that influenced how this approach evolved.

4. Chapter 5 presents an efficient GTM implementation for a multi-core CPU architecture which will be used as a baseline for comparison with the GPGPU algorithm.

5. Chapter 6 describes the experimental evaluation in terms of the data sets used, the timing methodology used, and the results observed.

6. Chapter 7 concludes the thesis, with a presentation of results and possible future work.

Chapter 2

Background

This chapter provides an overview of existing and established research for the topics discussed in this thesis, including an overview of text similarity and a description of the text similarity method selected to be accelerated in this thesis (the Google Trigram Method (GTM)). It also contains a review of the architecture and programming models for General Purpose Graphics Processing Units (GPGPUs).

2.1 Text Similarity

Text similarity, also known as document relatedness, plays a fundamentally important role in tasks such as information retrieval [5, 16, 18, 9], text classification [5, 16, 18, 9] and document clustering [5, 16, 18, 9, 6]. Accurately and effectively assessing document relatedness is increasingly critical as the volume of documents that can be compared is rapidly multiplying. Documents are increasingly digitized into repositories and other on-line collections, while new e-mail, blog and form posts are added to the vast collection of documents that can be analyzed on a daily basis.

A variety of approaches to the computation of document relatedness have been explored in existing literature [5, 7]. Some of the more common approaches are as follows:

1. String-Based Similarity Measures

   These measures compute similarity between two documents by measuring the similarity or dissimilarity (distance) between two documents (or strings) [5]. In general, these methods take only the pair of strings as inputs, and produce the relatedness between the strings using no other information other the strings' content. These distance methods can utilize the following approaches:

   (a) Edit Distance Measures [17]

Edit distance measures quantify the relatedness between two strings via a sum. This sum is produced from the number of modifications required to produce the first string from the contents of the second. These modifications can include insertions, deletions or substitutions.

(b) Bag Distance [17]

The bag distance approach enumerates all of the characters in the first string that cannot be matched with the characters of the second string. This approach then completes the opposite function, enumerating the characters from the second string that cannot be matched with the first. The maximum value of the enumerations is the bag distance.

(c) N-gram Measures [17]

N-gram measures count the number of n-grams, or substrings of length n that are common between the two strings [17]. Upon finding the number of common n-grams, the similarity between the two strings is determined by either dividing the number obtained by the number of n-grams in the shorter string, the larger string or by an average of the two strings. This formula is similar to the Dice Coefficient [8], which is twice the number of common characters in the compared strings divided by the total number of characters in both strings.

(d) Longest Common Substring (LCS) Measures [17, 5]

LCS is performed by determining the longest common substring between the compared strings. This longest substring is what will determine the overall relatedness of the two strings.

2. Knowledge-Based Similarity Measures [13, 5, 13]

These measures compute the text similarity of two documents by determining the relatedness between the words that make up the documents. This relatedness between words is retrieved from a knowledge-base provided for the approach. This knowledge base contains the semantic relatedness for word pairs, and in most knowledge-based measures, the knowledge base is provided via WordNet[4].

WordNet is a large lexical database for the English language, which groups nouns, verbs, adverbs and adjectives into structures called synsets. These synsets have defined relationships among the members of individual synsets as well as other synsets, and relatedness is calculated based on these relationships. These relationships are what is used to determine the relatedness of words which make up the documents.

3. Hybrid Measures [5]

   The hybrid measures make use of multiple similarity techniques to determine the relatedness between documents. One such example is Semantic Text Similarity (STS) [7], which determines the relatedness of two texts via string similarity and the semantic similarity, which is a knowledge measure.

4. Corpus-Based Similarity Measures [5, 9]

   Corpus-based similarity measures require two documents as inputs, as well as a collection of written texts, or corpus, to find the word relatedness. This corpus provides the necessary information regarding the relatedness between the words that compose the documents being compared. Document relatedness can then be produced using these inputs.

   An example of a corpus-based measure is the Google Trigram Method (GTM), presented in [9]. This approach uses the Google N-Gram Library, specifically the Uni-gram and Tri-gram corpora, to determine the relatedness between the words of two documents. Using the relatedness between all the word pairs between the two documents, the GTM produces a result indicating the relatedness between the documents. Additionally, this approach (proposed in [9]) could be used with any language, provided that there exists an n-gram corpus in that language.

After surveying the existing literature, it was decided to further investigate the acceleration of text relatedness performed using a corpus-based similarity measure, specifically the GTM. The GTM is an attractive approach because of the following factors:

1. The corpus required by the GTM, the Google N-gram corpus, is both free to use and publicly available.

2. The GTM provides the flexibility to perform relatedness calculations in languages other than English.

3. The GTM has reported superior performance over other word similarity measures in existing research[9].

4. The structure of the GTM appears to be amenable to parallelization.

2.2 Google Trigram Method (GTM)

The GTM was pioneered in [9] as an unsupervised approach to computing document relatedness. This is a corpus-based document similarity approach, which requires the following inputs to calculate document relatedness:

1. The two or more documents for which the relatedness is sought

2. The Google Web 1T N-Gram data set, as presented in [3]

The Google Web 1T N-Gram data collection provides five distinct n-gram data sets, those being uni-gram, bi-gram, tri-gram, quad-gram and quint-gram. Each of these data sets provide millions of n-grams and their associated rate of occurrence in Google web searches [3]. Of the 5 N-Gram data sets, the GTM requires only the tri-grams [8], and an accompanying uni-gram of the same language data set as well, though it does not need to be from the 5 N-Gram data sets.

The GTM uses the frequencies of the tri-grams to produce the relatedness between the words that make up the compared documents. The main tenant of the tri-gram relatedness model is to determine the relatedness between pairs of words, which are present in both the uni-gram and tri-gram corpora. This relatedness, or word similarity, is produced using the frequency of occurrence for the uni-grams and tri-grams that contain word pair as defined in [8], and is expressed in Equation 2.1.

$$Similarity(w_1, w_2) = \begin{cases} \frac{\log\left(\frac{(C(w_1,w_2)/2)C^2}{C(w_1)C(w_2)min(C(w_1),C(w_2))}\right)}{-2\log\frac{min(C(w_1),C(w_2))}{C}} & \text{if } \frac{(C(w_1,w_2)/2)C^2}{C(w_1)C(w_2)min(C(w_1),C(w_2))} > 1, \\ \frac{\log 1.0.1}{-2\log\frac{min(C(w_1),C(w_2))}{C}} & \text{if } \frac{(C(w_1,w_2)/2)C^2}{C(w_1)C(w_2)min(C(w_1),C(w_2))} < 1, \\ 0 & \text{if } (C(w_1,w_2)/2) = 0 \end{cases} \tag{2.1}$$

where: w_1 = first uni-gram from the tri-gram
w_2 = third uni-gram from the tri-gram
$C(w_i)$ = frequency for the given uni-gram
$C(w_1, w_2)$ = the frequency of tri-gram containing w_1 and w_2
C = maximum frequency in the uni-gram set

To perform the GTM, the documents and n-grams must have been read into memory. The GTM must then convert each document into a collection of tokens. The process of converting a given document into tokens is accomplished by removing all special characters and punctuation from the document. The tokens are created from only the words of the document. Once this process is completed, the document is now ready for use. Algorithm 1 describes how the GTM computes the relatedness between two documents, which is also known as calculating 1:1 relatedness.

To compute the relatedness between **N** documents in a corpus, or N:N relatedness, the GTM algorithm would be repeated over the corpus, alternating the documents used until each document in the corpus has been compared against all other documents. Given that document$_i$'s relatedness to document$_j$ is the same as document$_j$'s relatedness to document$_i$, the algorithm would need to be called $\frac{\mathbf{N}^2}{2}$ times to produce the result. The relatedness values between a given document pair (i,j), if placed in a matrix, would result in an upper triangular matrix as seen in Figure 2.1.

$$Mean = \frac{\sum_{c=0}^{N} WRM(w_{row}, w_c)}{N} \tag{2.2}$$

where: N = the length of document$_2$ after removal of matching words
w_{row} = the rowth element of document$_1$
w_c = the columnth element of document$_2$
$WRM(w_r, w_c)$ = the word relatedness between the two words

$$StandardDeviation = \sqrt{\frac{\sum_{c=0}^{N} (WRM(w_{row}, w_c) - U)^2}{N}} \tag{2.3}$$

Algorithm 1 1:1 GTM Text Relatedness

Require:

Document$_1$: A document that has been converted to tokens. It is a sequence of $|Document_1|$ words

Document$_2$: A document that has been converted to tokens. It is a sequence of $|Document_2|$ words

Google Web 1-Gram data set

Google Web 3-Gram data set

Ensure: $|Document_1| \leq |Document_2|$, if this not the case then switch the contents of Document$_1$ and Document$_2$

Step 1: Remove and count number of words that occur in both documents. If all of the words match, proceed to **Step 5**.

Step 2: Construct a matrix, called the **WRM**, where the value stored at row **r** and column **c** is the word relatedness of $\mathbf{r}^{th}$ word of document$_1$ and the $\mathbf{c}^{th}$ word of document$_2$ as illustrated in Figure 2.2. The word relatedness values are calculated as per Equation 2.1, using the frequencies found when searching for $\mathbf{r}^{th}$ word of document$_1$ and the $\mathbf{c}^{th}$ word of document$_2$ in the tri-gram and uni-gram data sets.

Step 3: For each row **r**, find the mean using Equation 2.2. Once the mean is known, find the standard deviation of the same row using Equation 2.3. Once these values have been computed, store the summation of the mean and standard deviation of each row as a value called $\mathbf{K}_r$

Step 4: For each row **r**, create the set of values in which the index is $\geq \mathbf{K}_r$, called $\mathbf{A}_r$. Upon processing of the entire row, find the mean of $\mathbf{A}_r$ and add it to the value **X**.

Step 5: Determine the relatedness between the two documents using Equation 2.5, which returns a value between 0 and 1.

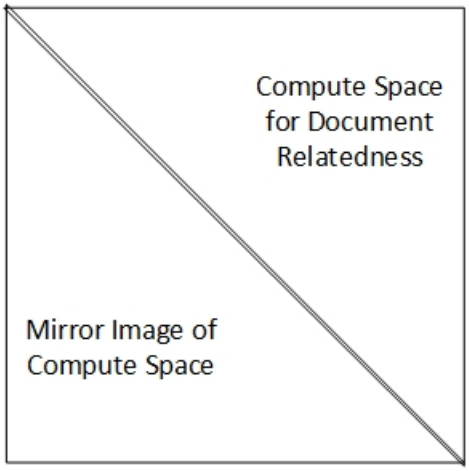

Figure 2.1: Representation of the Calculation Space Document Relatedness

where: N = the length of document$_2$ after removal of matching words
U = the mean of the row
w_{row} = the rowth element document$_1$
w_c = the columnth element of document$_2$
$WRM(w_r, w_c)$ = the word relatedness between the two words

$$X = \sum_{r=0}^{M} \frac{A_r}{|A_r|} \tag{2.4}$$

where: M = the length of document$_1$ after removal of matching words
A_r = the elements of the rth row $\geq$ mean$_r$+standard deviation$_r$

$$DocumentRelatedness = \frac{(Y + X)(AB)}{2AB} \tag{2.5}$$

where: Y = number of matching words between Document$_1$ and Document$_2$
X = summation of all the significant word relatedness
A = the length of document$_1$
B = the length of document$_2$

Word Similarity Between Pair: $Doc1r_0, Doc2c_0$	Word Similarity Between Pair: $Doc1r_0, Doc2c_1$	Word Similarity Between Pair: $Doc1r_0, Doc2c_2$	Word Similarity Between Pair: $Doc1r_0, Doc2c_3$
Word Similarity Between Pair: $Doc1r_1, Doc2c_0$	Word Similarity Between Pair: $Doc1r_1, Doc2c_1$	Word Similarity Between Pair: $Doc1r_1, Doc2c_2$	Word Similarity Between Pair: $Doc1r_1, Doc2c_3$
Word Similarity Between Pair: $Doc1r_2, Doc2c_0$	Word Similarity Between Pair: $Doc1r_2, Doc2c_1$	Word Similarity Between Pair: $Doc1r_2, Doc2c_2$	Word Similarity Between Pair: $Doc1r_2, Doc2c_3$

Figure 2.2: Example of the Matrix constructed from Document_1 and Document_2 following Step 2 from Algorithm 1

2.3 General Purpose Graphics Processing Unit Computing

General Purpose Graphics Processing Units (GPGPUs) make use of a computer's Graphics Processing Unit (GPU), a specialized add-on card designed to rapidly manipulate and create images for a computer's display, in conjunction with the computer's CPU to accelerate applications. Any observed increase in performance depends on how effectively the application can offload the computations from the CPU to the GPU, and how efficiently the applications can be mapped to the specialized GPGPU architecture.

This mapping of an application requires that it be reworked from the classic CPU design of sequential execution of tasks, to that of a GPGPU, which utilizes data-parallel execution of tasks. This data-parallel execution stems from the GPU's evolution to update the computer's display without noticeable artifacts. To reduce or eliminate display issues, the GPU hardware has evolved to execute the same instructions on different elements of data, or to display information.

The specific GPU selected for use in the GPGPU computing presented in this thesis is a Nvidia GPU. This hardware was selected for its performance over traditional CPU architectures as illustrated in Figure 2.3, the ongoing support and development of CUDA, and near dominance of the add-in GPU market [1]. As such, the information that follows will focus on and be tailored towards Nvidia's GPUs.

2.3.1 History of GPGPU Computing

The evolution of GPUs from a specialized chipset that performed display updates to a general purpose computing hardware was not spurred by the foresight of graphics card manufacturers. It was in fact due to Microsofts requirement that all DirectX

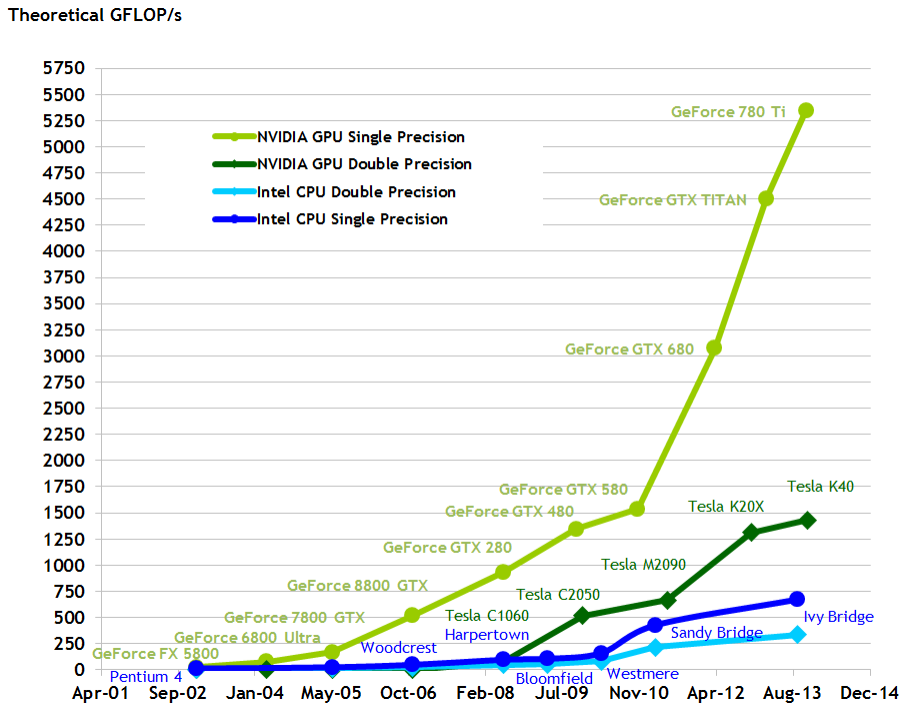

Figure 2.3: Contrasting the CPU and Nvidia GPU Architecture Performance in Terms of GFLOPS

8.0 compatible devices include programmable vertex and pixel shaders to improve 3D graphics rendering performance.

The GeForce 3, a GPU produced by Nvidia, was the first to meet the Direct X 8.0 standard and thus was the first card to provide programmers the ability to exact a degree of control over what calculations would be performed on the GPU. Early attempts to use GPUs for generalized computing required that the developer present their work as a series of OpenGL or DirectX API calls. This mandated that GPGPU tasks must appear to the GPU as rendering tasks, thus limiting the usefulness and increasing the complexity of leveraging the hardware. Based on the complexity of utilizing these computing resources, and the limitations of the Direct X framework, basic computing tasks such as random reads and writes to memory were difficult, if not impossible, to perform. This meant that programs that required scatter and gather operations, the sending and receiving of data from node to the all the others, were unsuited to application on GPGPUs.

Nvidia hardware navigated these challenges until the release of the Geforce 8800 GTX in support of DirectX 10, which supported Microsofts latest revision of the 3D

standard, and would be the first GPU built to support Nvidia's Compute Unified Device Architecture (CUDA). CUDA's architecture was designed to facilitate general purpose computation by moving away from the previous models of vertex and pixel shaders, which older versions of DirectX required, and instead focusing on a unified shader pipeline, which DirectX 10 used. This unified shader pipeline could be used for more generalized tasks, however, it required the tasks to be expressed in either Microsoft's High Level Shader Language(HLSL) or OpenGL's Shading Language (GLSL).

It was not until Nvidia released the CUDA C programming language that Nvidia's GPU could easily be utilized to perform general purpose computing without the need to understand Direct X or OpenGL.

2.3.2 Nvidia's GPGPU Architecture

Nvidia has produced several iterations of GPGPU hardware architectures, all of which are based on CUDA, and has revised them approximately every 2 years. These architectures are:

1. Fermi, released in 2010

2. Kepler, released in 2012

3. Maxwell, released in 2014

4. Pascal, scheduled for release in 2016

This thesis makes use of a Kepler-based GeForce Series GPGPU, as this was the most recent and readily available architecture upon commencement of this research in late 2013. Figure 2.4, illustrates the average performance of Nvidia GeForce Series GPU architectures in terms of GFLOPS. These values are drawn from the performance of each GeForce Series GPU that adheres to the given architecture for that calendar year. The mid-level Kepler card used in this thesis, a GeForce GTX 660, is expected to have a peak performance of 2,100 GFLOPS.

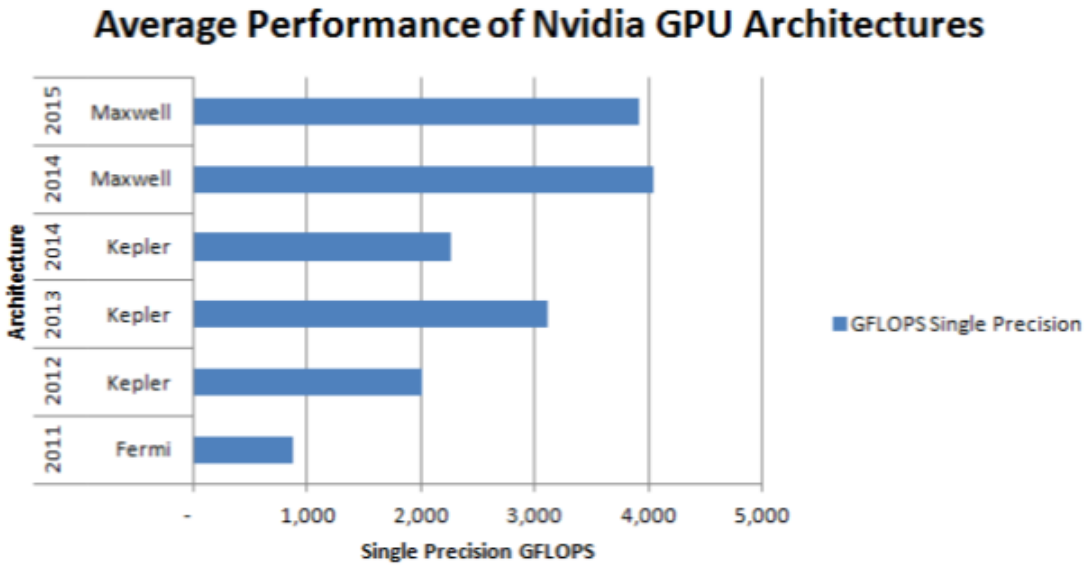

Figure 2.4: Average Nvidia GPU Architecture Performance as Observed Across all GeForce Series Cards

2.3.3 CUDA Model

In CUDA, any task performed on the GPU is composed of threads, and the processors of the GPU are focused on executing those threads. These processors are organized into Streaming Multiprocessors (SMs)[1] and are composed of CUDA cores and other execution units. The SMs group the assigned threads into groups of thirty-two, referred to as a warp, and then execute the warps as needed. Note that in general, the CUDA programming model is organized in such a way that the programmer does not need to be concerned with warp allocation, and instead can write the application code from the perspective of a single thread.

While the programmer is shielded from the low-level resource assignment, they are not protected from the execution of the hardware. Unlike traditional multi-core CPUs, which are Multiple Instruction/Multiple Data, the CUDA hardware and software models make use of the Single Instruction Multiple Threads/Data (SIMT/SIMD) architectures. A SIMT/SIMD architecture is one in which only a single instruction is executed at a time, however it is applied to multiple data points simultaneously. The classic example of a SIMD task would be adding a numeric to each element in a vector. The single instruction would be to add the numeric to the value stored

[1] In the Kepler card used in this thesis, that number is five

at vector index, however, this addition would be performed on all elements in the vector at once. In a CUDA GPGPU, SIMT/SIMD is represented by the processing elements. At the software level this is a thread, and at the hardware level it is the work being performed by a CUDA core, that would perform the same operation on differing locations in memory.

As mentioned previously, CUDA programs can be written from the perspective of a single thread, however, the code contained there-in applies to all threads that will run that section of the program. As the CUDA is a SIMT/SIMD architecture, the programmer would want to ensure that the instructions being processed can apply to all threads. In the case of conditional branching, performance can degrade if the threads are no longer processing the same data. This occurrence is called warp divergence, and should be avoided to the greatest extent possible as the threads are not all performing the same task, and are therefore slowing the overall execution.

2.3.4 CUDA C

Nvidia provides the programming language, CUDA C, to write applications that will run on their GPUs. As the name implies, CUDA C is very similar to C both in syntax and structure. A CUDA program is composed of one or more kernels, similar in concept to C-style functions, that are provided GPU resources based on launch parameters. Launch parameters define the number of threads to allocate to the kernel and what resources are made available to the threads. The thread allocation is two-fold. First is the number of blocks, or thread pools requested, and second is the number of threads in a given block. These resources can be mapped into a three dimensional (X,Y,Z) co-ordinate system called a grid. For example, if a kernel specified as follows:

1. exampleKernel <<<new Grid(3,2,0),new Grid(3,3,3)>>>

it is assigned device resources as illustrated in Figure 2.5.

Using the above kernel as our exemplar, the kernel is executed on the concurrent parallel thread pools. Each thread within the pool executes on an instance of the kernel, with each thread aware of its unique identifier within the pool and the ID of the block to which they are a member. Additionally, the threads of a given pool can

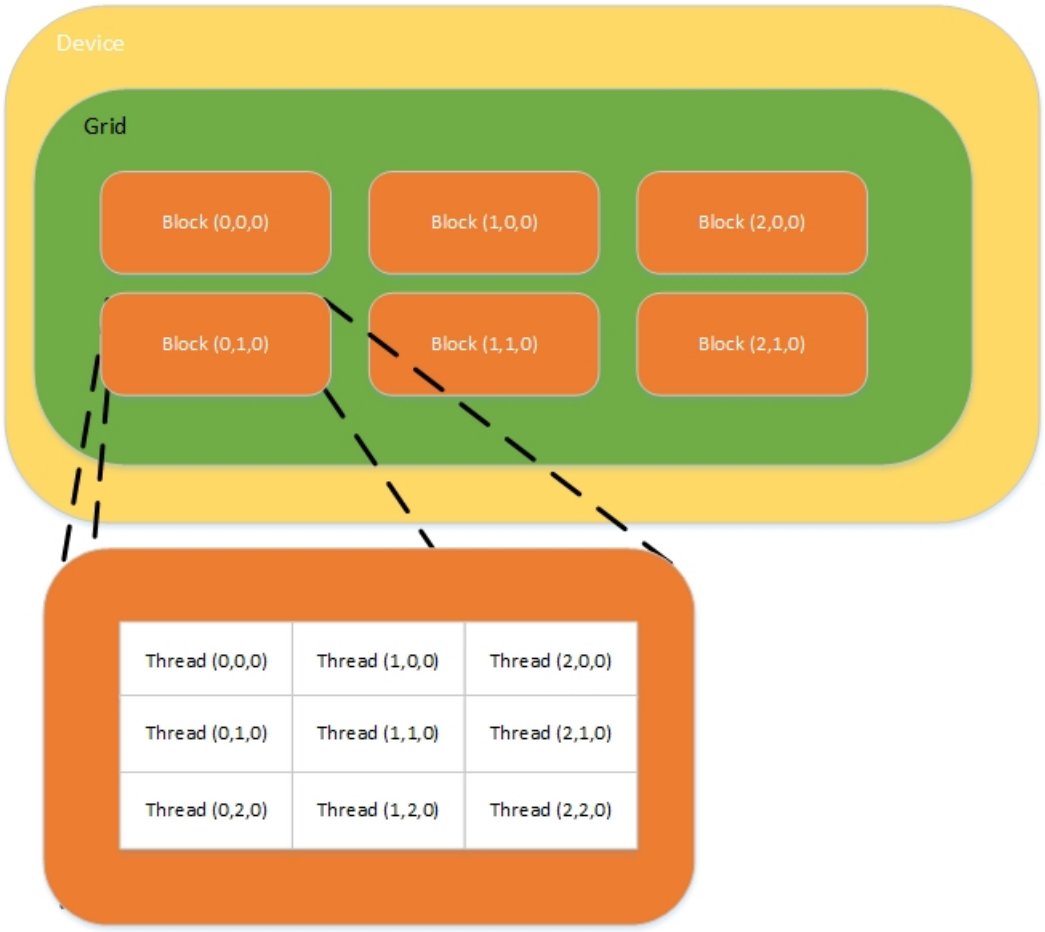

Figure 2.5: Device Kernel Allocation

co-operate with each other via shared memory and through synchronization (barriers). A thread is provided with its own local memory, access to the shared memory, and access to read and write to global memory.

This memory hierarchy is captured in Figure 2.6, and it should be noted that the Read Only Cache is a tunable feature, which is carved out of the available shared memory, and is not used in this thesis[2]. The memory model presented is available to the GPGPU when a kernel is being executed. Otherwise, outside of the kernel calls, the GPGPU only exposes its DRAM or global memory to the developer. This allows the developer to allocate memory on the GPGPU for data structures and other elements required to perform the computations in the kernels. Additionally, it allows

[2]This thesis was implemented to work on all CUDA devices, thus the assumption will be a Compute 1.0 device, which would not have Read Only Cache.

for the movement of data between CPU memory and the GPGPU.

This facilitates the movement of data between the CPU memory and the GPGPU.

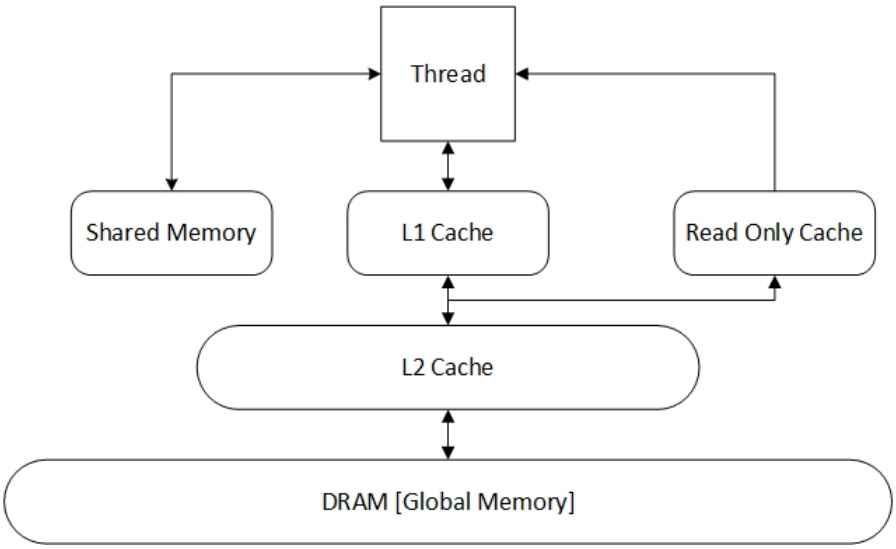

Figure 2.6: Nvidia GPU Memory Hierarchy

2.3.5 Efficiency Concerns in GPGPU Programming

When programming for the GPGPU, there are several key factors to keep in mind. As mentioned in Section 2.3.3, the underlying architectural model of the GPGPU is SIMT/SIMD, and therefore, conditional branching should be avoided as much as possible. In the cases where branching cannot be avoided, it is better to structure the program in such a way that reduces the scope of the branching. For example, if a branch is used to determine the success of a look-up in a data structure and the element is not found, the branch should assign a value that will have no impact on future operations. This reduces the scope of the branch's divergence in execution of the threads, rather than dealing with the case of missed look-ups throughout the rest of the program.

In Figure 2.6, we see that the GPU has three layers of memory, each of which provides access that is slower than the last. A GPGPU application should be coded to minimally access global memory, performing as much work as possible in the L1 Cache or shared memory. In a best case scenario, the kernel should read global

memory only at kernel launch, and when writing the results back.

Another consideration is to reduce the amount of data copied to and from the GPU as the data transfer between the GPU and CPU is an expensive operation. Adhering to the distributed system design of small infrequent messaging would be the best practice.

With these considerations in mind, this thesis will attempt to create an approach for applying the GTM relatedness computation for the CPU to the GPGPU. Based on the structure of the algorithm presented in Algorithm 1, the algorithm can be expressed as a series of parallel computations, for example, Steps 2 through 4. Additionally, it requires no data is fetched from memory other than the values that are derived from the WRM. Section 3 will outline how the GPGPU approach is to be developed.

Chapter 3

Framework for Implementing of the GTM on a GPGPU

This chapter explains the design choices and rationale behind the foundational elements of the implementation of the GTM on a GPGPU. The design choices were selected because they contributed to an efficient solution for applying the GTM to GPGPUs. This approach, including the concepts of pre-computing word similarities and tokenizing documents, draws from the existing body of work presented in [15]. However, this approach differs in both the applied data structures, and in the approach and performance of calculating N:N document relatedness.

3.1 Design Assumptions and Influences

During the development of the approaches to be examined in this thesis, the following core design choices were made:

1. Pre-computation will be utilized wherever possible.

2. All document relatedness processing will be done in memory to the greatest extent possible.

3. The workstation which houses the GPGPU will have sufficient memory to hold all N documents to be compared on the GPGPU.

Pre-computation is the concept of performing applicable work in advance, such as processing the documents of the corpora, or creating the Word Relatedness Matrix (WRM). Work that can be pre-computed is typically work that will not change at run-time or be affected by any computations, and therefore one can save resources during the computation by completing this work in advance. This will benefit future tasks which make use of common outputs to perform other functions, as there will be no additional costs to begin processing.

Making the most of a targeted hardware's system memory is a common engineering strategy, and accordingly, any processing should be done in memory wherever possible. This helped inform the choice to use pre-computation wherever possible, meaning that pre-computed values should be stored in memory as close to the processing cores as possible. The WRM and the documents of the corpus that will be compared should be read from storage and into CPU memory prior to performing document relatedness calculations.

The third design assumption is that the memory attached to the CPU will have sufficient space to hold all of the processed documents that will be compared. As this thesis leverages GPGPUs, which maintain separate memory from the CPU, this assumption allows us to focus on devising an effective method for transferring the documents stored in the CPU memory to the GPGPU memory. The CPU to GPGPU memory allocation methods will be discussed in Section 4.5.

3.2 Design Considerations

Prior to introducing the high-level design for the tools applied in this thesis, a brief overview of the relevant design considerations is necessary.

In this thesis it is essential to reduce the space required for the inputs measured by the document relatedness computation as it will be performed on a GPGPU. To this end, the decision was made to tokenize character strings into unique numeric values. This will often decrease the memory space required for storage and reduce the time needed for future operations, such as equality checks, when compared to a non-tokenized format.

This thesis also gave careful consideration to how the WRM would be represented. Historically, the WRM is produced by determining the relatedness between each word in a uni-gram set, and all the other words present in the set. This leads to a two-dimensional array (matrix), which holds the relatedness between any given pair of words. The WRM is indexed in row-column order, therefore allowing the relatedness between two words to be indexed by WRM_{ij}, or WRM_{ji}.

If V is the size of the uni-gram set used, then the WRM would contain V^2 elements. For smaller vocabularies, this representation of the WRM will fit in memory, but for larger vocabularies the WRM quickly becomes too great for memory. Given that

GPGPU memory sizes are considerably smaller than those of CPUs, a more space-efficient version of the WRM is required for this thesis in order to design an approach applicable to larger computations of relatedness.

To resolve this issue, the content and nature of the WRM was examined. As mentioned previously, the WRM provides two distinct indices that point to the same value, WRM_{ij} and WRM_{ji}. Additionally, not all word pairs will have a non-zero relatedness value, leading to a matrix that is sparsely populated with non-zero values. To address this, we can modify the WRM to hold the same amount of information while requiring significantly less space.

This optimization is accomplished by organizing the WRM so that the value at WRM_{ij} or WRM_{ji} is indexed by the smallest value of the (i,j) pair. This allows for the total space of the WRM to be halved, but will require that the WRM is remapped to take advantage of the reduction. During the process of performing this re-organizing, the indicies which have a relatedness value of zero can be excluded. Figure 3.1 illustrates this process.

	Word 1	Word 2	Word 3	Word 4
Word 1	1	.3	.5	.6
Word 2	.3	0	.4	.8
Word 3	.5	.4	.5	.9
Word 4	.6	.8	.9	0

(a)

	Word 1	Word 2	Word 3	Word 4
Word 1	1	.3	.5	.6
Word 2			.4	.8
Word 3			.5	.9
Word 4				

(b)

Figure 3.1: Remapping the WRM to provide non-redundant access. (a) Illustrates the WRM prior to removing redundant access, and zero values. (b) Illustrates the transitory state of the WRM once zero values, and sorted access is in place.

The final step in the alteration of the WRM remaps the matrix into a 1D array. This process removes the ability to easily access the WRM via direct look-ups and requires additional information, such as the word pair to which the relatedness value relates. Figure 3.2 illustrates the resulting data structure to hold the WRM.

This simplified form of the WRM is more space efficient, but the data contained therein is no longer as easily accessible, resulting in the need for look-up strategies.

Word 1	Word 1	Word 1	Word 1	Word 2	Word 2	Word 3	Word 4
Word 1	Word 2	Word 3	Word 4	Word 3	Word 4	Word 3	Word 3
1	.3	.5	.6	.4	.8	0.5	.9

Figure 3.2: The remapped WRM

The data structures required will be determined based on the target hardware performing document relatedness computations, and thus each of these implementation decisions are considered for each unique architecture. This will be discussed in detail in Section 4.4.3 for the GPGPU approach and Section 5.2 for the CPU approach.

3.3 System Design

Prior to calculating document relatedness, the following processing steps must be undertaken:

1. The N-grams must be tokenized.
2. The documents of the corpus that are going to be used in document relatedness calculations must be tokenized.
3. The Word Relatedness Matrix (WRM) must be constructed from the tokenized N-Grams.

This sequence of steps is encapsulated in the high-level design for the proposed framework in Figure 3.3. As the figure illustrates, the design of the system is bifurcated into two separate functional areas. The CPU functional area is responsible for the processing of the N-Grams and the corpora as well as providing a baseline measure. This baseline measure for document relatedness is used to determine the effectiveness of GPGPU approach. The GPGPU functional area is responsible only for calculating document relatedness.

3.3.1 Design of the N-Gram Processor Tools

As previously mentioned, the N-Grams used in the construction of the WRM must be tokenized prior to being used in document relatedness calculations. Figure 3.4

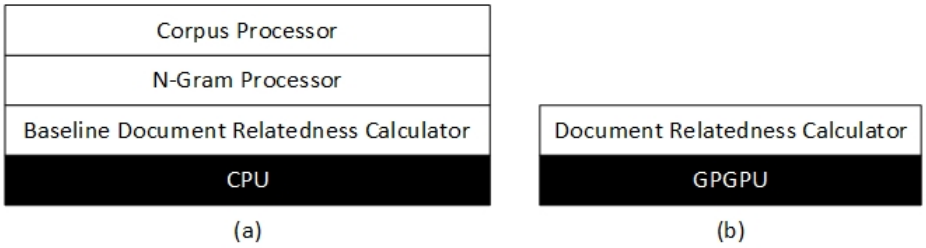

Figure 3.3: Proposed System Design for Implementing GTM. (a) Indicates that the processing of Corpora and N-Grams are to targeted to run on CPUs and provide of measure of performance for document relatedness calculations, while (b) indicates that the Document Relatedness routines are to be targeted to run GPGPUs

illustrates the high-level of the N-Gram processor. This system is composed of three tools that, when used in series, are responsible for manipulation of the N-Grams into a WRM, which is required for document relatedness calculations.

The processing of uni-grams and tri-grams needs to only be performed once per N-Gram data set, and can be performed in any order, however, for the purpose of clarity they will be discussed in the order presented in Figure 3.4.

After processing the uni-grams and tri-grams, the WRM can be created and saved to storage for future use in the document relatedness calculations. The details of how the WRM is created, will be outlined in Section 3.5.

3.3.2 Design of the Corpus Processor Tool

Figure 3.5 illustrates the high-level design of the tool that converts each document of the ASCII formatted corpus into a processed document that can be used by the Document Relatedness Module.

3.3.3 Design of the Document Relatedness Tool

Figure 3.6 shows the high-level block design of the document relatedness tool.

Prior to exploring the inputs required for the tool to function, we must discuss the algorithm used to calculate the document relatedness, and how this algorithm can be mapped to both the GPGPU and CPU.

The algorithm used to determine document relatedness is common to both the CPU and GPGPU, regardless of the target architecture. This algorithm is presented

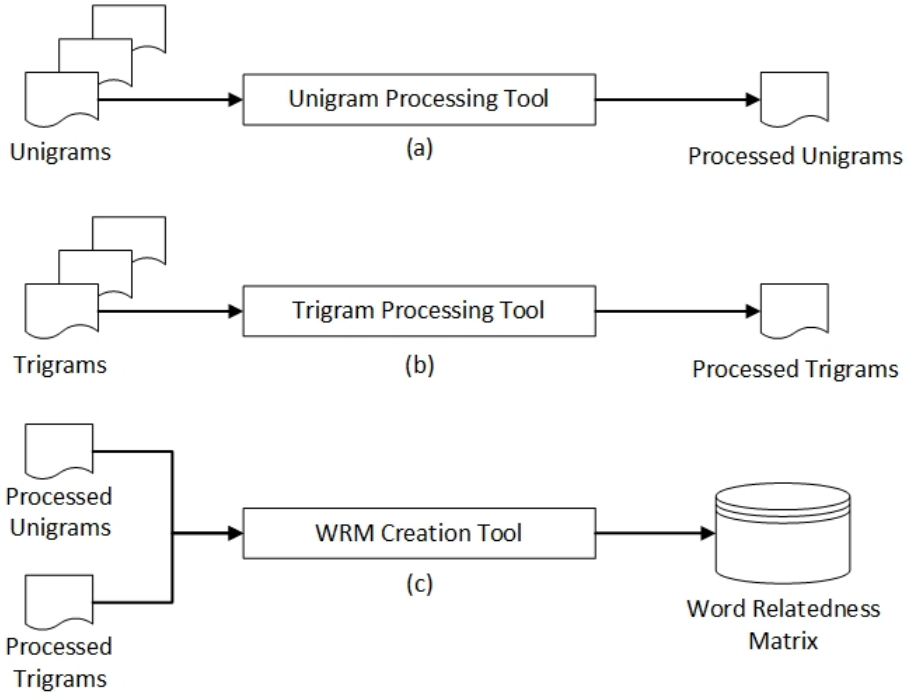

Figure 3.4: N-Gram Processor Block Diagrams for the GTM Framework. (a) Illustrates the Uni-gram Processing Data Flow, (b) Illustrates the Tri-gram Data Flow, and (c) Illustrates the WRM Data Flow.

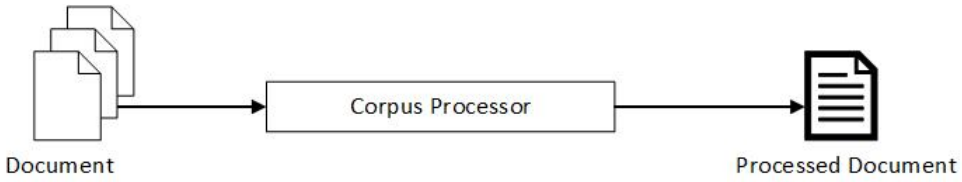

Figure 3.5: Corpus Processor Block Diagram for the GTM Framework.

in Algorithm 1.

The minimum pre-computational work that each approach requires is a WRM and a pair of processed documents. Provided these inputs, the document relatedness tool will produce an output between 0..1 representing the relatedness between the pair of documents as determined by Algorithm 1. However, this tool is designed and implemented to compare more than just a single pair of documents, and also aims to perform the comparison in the most efficient manner.

The design implementation rationale as well as the performance of the discussed representation of the WRM will be discussed for the GPGPU in Chapter 4, and for

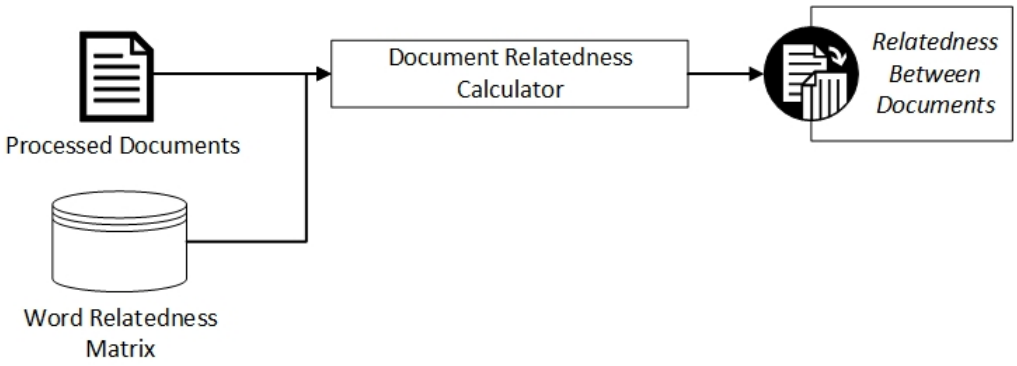

Figure 3.6: Document Relatedness Calculator Block Diagram for the GTM Framework.

the CPU in Chapter 5.

3.4 Generalized N-Gram Processing

The N-gram corpus, regardless of source, presents each entry in the corpus as a delimited sequence of N words and a count representing the occurrence of that given N-gram. In the case of the N-gram used in this thesis, the Google Web 1T N-Gram corpus, the counts are the occurrence that the given N-gram appeared in one trillion English web texts [3].

In this thesis we will tokenize each of the words that compose an N-gram, and will represent each of the tokenized words with their own unique ID, called a *WordID*.

The tokenization process requires that each N-Gram data set is read independently and sequentially. As each entry is read, the end of an N-gram is determined by the delimiters for the data set, generally a whitespace character such as a tab, and the end of the entry itself via a new line. Each entry in the N-gram data set is treated as a series of N words with a single numeric entry.

As each sequence of the N-gram is read, each word of the N-gram is converted to lower case, and checked to confirm that it is composed only of alphabetic characters. If non-alphabetic values occur in any of the words of the N-gram, the whole N-gram is discarded as invalid. If the word is valid it is given a *WordID*.

This *WordID* can be assigned via the Dictionary Look-Up approach. The Dictionary Look-Up approach is a mapping technique leveraged in previous works [19, 15]

to accelerate the GTM. The Dictionary Look-Up approach first requires that all N-grams be read into memory. Once in memory, only the unique words are placed in a word table. The words in the word table are then assigned a unique value, from 0 to the number of unique words-1, based on the order that they were read from the corpus.

Adding a new word to vocabulary requires that the word table is scanned to determine if the word is unique, in which case it is added to the table. As the number of unique words increases, the space required for the word table grows, as does the number of accesses required to determine the *WordID* for a given word.

If the N-gram has been read, and is valid, it is stored in memory in the following format:

1. N *WordID*s
2. Count
3. Frequency

In the case of duplicate N-grams, (N-grams that have the same *WordID[s]*) the existing N-gram is kept in memory, but has its count increased to account for the existence of a duplicate N-gram, and the duplicate is then discarded. Once all of the N-grams have been read from disk, each one has its frequency derived, as per Equation 3.1.

$$Frequency = \frac{O}{\sum_{i=0}^{M} C_i} \tag{3.1}$$

where: M = the number N-grams in the data set
C_i = the count value of the ith N-Gram
O = the count of the N-gram for which the frequency is to be calculated

Upon completion of this calculation for all of the N-grams in memory, the N-grams are written to disk for later use.

This generalized structure holds true for uni-grams, however, it does not for tri-grams. Based on the work presented in [9], a tri-gram consisting of word$_1$, word$_i$, and word$_3$ can be combined with any tri-gram that also consists of word$_1$ and word$_3$ in the same position. Further, a tri-gram consisting of word$_1$, word$_i$, and word$_3$ can also

be combined with the tri-gram consisting of $word_3$, $word_i$, and $word_1$, provided that the mean frequency for the two tri-grams is used.

Therefore, each element of the tri-gram data set written to disk is represented as:

1. $WordID_1$
2. $WordID_2$
3. Count
4. Frequency

This applies when the entries are sorted such that $WordID_1 \leq WordID_2$, and the frequency is half of the value of the result from Equation 3.1 using the given count.

3.5 Word Relatedness Matrix

The WRM Creation Tool produces a WRM in the format described in Section 3.2.

Each of the words that compose a tri-gram, $WordID_1$ and $WordID_2$, if found in the uni-gram data set, have the required values from the uni-gram data set as well as those of the tri-gram data set provided for Equation 2.1. The resulting similarity between $WordID_1$ and $WordID_2$ is placed into the WRM associated with whichever has the smallest *WordID*. Once all of the tri-grams have been evaluated, the reduced WRM is saved to disk for future use.

3.6 Corpus Processing

The Corpus Processor processes the provided corpus one document at a time. Each of the documents processed are treated as a collection of words delimited by punctuation and whitespace. Similar to the N-gram processing, each word is converted to lowercase and checked to confirm it is a strictly alphabetic sequence. If the word only consists of alphabetic characters, it is deemed valid and given a *WordID*, and is stored in memory until the document has finished processing. If the word is invalid it is discarded. In the case of duplicate *WordIDs*, only the first occurrence of the word is kept. Once all of the words in the document have been processed, the processed document is then written to disk storage for future use in document relatedness calculations.

Chapter 4

Computing GTM Document Relatedness on GPGPUs

This chapter explains the design details for the GPGPU implementation of the document relatedness approaches from Chapter 3.

The generalized algorithm used to calculate document relatedness from Section 2.2 is mapped to the GPGPU hardware in several different ways. After the creation of an algorithmic approach, the data structures used to support it are investigated and evaluated and subsequently the methods for transferring documents from the CPU to the GPGPU are analyzed.

This investigation and analysis generated two possible approaches: the global memory approach and shared memory approach. Both will be explored throughout this analysis.

4.1 GPGPU Text Relatedness Algorithm and Mapping it to the GPGPU Architecture

The generalized algorithm presented in Section 2.2 can be adapted to the GPGPU architecture with modifications as shown in Algorithm 2. Note that some of lines in Algorithm 2 correspond to the GTM steps from Algorithm 1, while others are just scaffolding required by the GPGPU method.

This thesis leverages Nvidia GPGPUs and implements the algorithm in CUDA C. When Algorithm 2 is mapped to the GPGPU hardware, CUDA terminology and limitations are applied to outline the design and implementation specifics. An additional consideration is the disparate memory structures of the CPU and GPGPU platforms. Consequently, the approach to copying any documents to the GPGPU is an added complexity that must be properly managed, both due to the cost of transferring documents into GPGPU memory and the scarcity of memory. Once documents are transferred to the GPGPU, document usage must be fully maximized in order to reduce the number of memory allocations and/or memory copies required. This is

Algorithm 2 High Level Description of the GPGPU Approach

Require: The corpus and WRM are pre-processed. The user provides a set of documents to be compared

1: $wrm \leftarrow$ READ WRM
2: $corpus \leftarrow$ READ CORPUS
3: Copy the wrm and the $corpus$ to the GPGPU
4: On the GPGPU perform the following:
5: **for** $i \leftarrow 0; i < |Corpus|; i++$ **do**
6: $doc1 \leftarrow Corpus_i$
7: **for** $j \leftarrow i; j < |Corpus|; j++$ **do**
8: $doc2 \leftarrow Corpus_j$
9: /*GTM Step 1 is done in Flag Matches*/
10: $numberOfmatches \leftarrow$ FLAG MATCHES$(doc1, doc2)$
11: /*GTM Step 2 to 4 is done in Calculate Relatedness*/
12: $relatedness \leftarrow$ CALCULATE RELATEDNESS$(doc1, doc2, wrm)$
13: Copy the $relatedness$ from the GPGPU to the CPU
14: Copy the $numberOfmatches$ from the GPGPU to the CPU
15: Perform the final relatedness calculation on CPU /*GTM Step 5*/
16: Record the final relatedness to disk
17: **end for**
18: **end for**

accomplished by flagging the words that match in a pair of documents rather than removing those matching elements from the documents. This approach allows for document re-use on the GPGPU provided that those flags are cleared, and relevant elements are excluded from the calculation of relatedness. By checking for matching words in the documents prior to calculating relatedness, we can save the overhead of launching a potentially unnecessary and more complex kernel that would require additional hardware resources.

The first step in determining how to map the algorithm to the GPGPU hardware is to identify where the bulk of the computational work is performed in Algorithm 2. Then we must find a method to effectively express the results on a Nvidia GPGPU, within the limitations of the CUDA framework.

Step 2 through 4 of Algorithm 1 all require construction of a matrix from the selected documents, which will be used to perform various operations. Mapping of the algorithm can begin, using the construction of the matrix as a starting point. As only the number of blocks and threads per block have to be specified in each function call, $|document_i|$ and $|document_j|$ are provided as parameters.

Note that for the remainder of this chapter, unless otherwise noted, it is assumed that $|document_i| \leq |document_j|$

4.1.1 Evolution of the Pair-Wise Approach

While Algorithm 1 provides a high-level description of how to perform the GTM, Algorithm 3 provides more detail for how the GTM would be implemented in code.

By examining Algorithm 3, we observe that work performed using the GTM is driven by the two inner 'for' loops. Specifically, it is centered around the look-ups for similarity between the words of the two documents, and work performed on these values as discussed Step 3 through 4 in Algorithm 1.

By organizing our GPGPU approach around performing 1:1 document relatedness as described in the GTM algorithms, the following mapping as illustrated in Figure 4.1 was selected.

Figure 4.1 illustrates that the mapping of a pair of documents for which relatedness was being calculated was derived from the two inner 'for' loops in Algorithm 3. By making the blocks equal to the $|document_i|$, and the number of threads per block equal

Algorithm 3 GTM Algorithm Sketch

```
for i ← 0; i < |Corpus| ; i++ do
    for j ← j; j < |Corpus| ; j++ do
        for k ← 0; k < |Document_i| ; k++ do
            for l ← 0; l < |Document_j| ; l++ do
                Compute Similarity between k, l
            end for
            Compute mean, standard deviation ....
        end for
        Write/Display Document i, j Similarity
    end for
end for
```

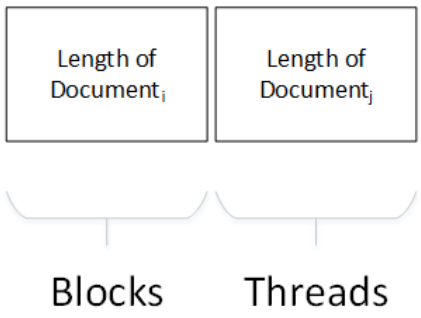

Figure 4.1: GPGPU Pair-Wise Approach Mapping to Kernel Parameters

to $|document_j|$, each block can perform all of the required work for one iteration of the third 'for' loop. Specifically, this allows for the work required for each row_r, Step 2 through 4 from Algorithm 1, to be performed by a single block.

Upon completion of the Pair-Wise kernel's execution, each row's $\mathbf{A}_r$ is known. An additional sub-step is required to perform a reduction on the results, which will produce the final output of GTM Step Four. This value can then be used in Equation 2.5 as the value of $\mathbf{X}$.

To determine and remove the number of matching words between the two documents being compared, an identical mapping to Figure 4.1 will be used. By mapping $|document_i|$ once more to the number of blocks, and $|document_j|$ to the number of threads per block, each pool of threads will determine if a single word in document_i matches any of the words in document_j. These words will be flagged as a match in both documents and a count of the number of matches found per row is generated. The number of matches are then reduced to a single value, which is used as $\mathbf{Y}$ in Equation 2.5.

The resulting Pair-Wise approach requires four CUDA kernel calls to calculate the relatedness between any two documents, namely:

1. Find matching words between the two documents
2. Perform a reduction on the count of matches
3. Calculate relatedness of each row
4. Perform a reduction on the relatedness of all rows

The values of $\mathbf{X}$ and $\mathbf{Y}$ are copied back to the CPU, which performs Equation 2.5. This calculation is performed on the CPU rather than the GPGPU because the GPGPU does not have the means to effectively perform it in parallel.

To evaluate the performance of this approach, the number of accesses into the GPGPU's global memory are examined, as this constitutes one of the most expensive operations that the GPGPU can perform.

Due to the Pair-Wise approach being organized by the number of blocks and the number of threads per block, a performance of $\mathbf{O}(NM)$ global memory accesses is expected if $\mathbf{N}$ is $|document_i|$ and $\mathbf{M}$ is $|document_j|$.

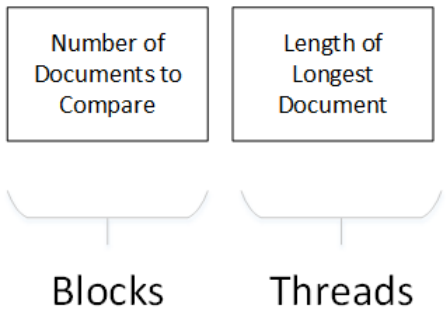

Figure 4.2: GPGPU One-To-N Approach Mapping to Kernel Parameters

4.1.2 Evolution of the One-to-N Approach

It is possible to achieve $\mathbf{O}(N + M)$ global memory accesses per pair of documents evaluated by developing a One-to-N approach, which makes heavier use of shared memory between the threads.

Figure 4.2 illustrates that the One-to-N approach is achieved by mapping the blocks to the number of documents to be compared at once, and the number threads to the longest document of the documents to be compared. Rather than one CUDA kernel per pair of documents $(document_i, document_j)$, there can now be several pairs of documents compared for a single kernel launch, $(document_i, document_*)$, where * is the number of blocks.

As observed in the Pair-Wise approach, the threads will read the longest document into thread memory. However, as there is only one block assigned per pair of documents, the threads must then iterate over each element of $document_{smallest}$, as illustrated in Figure 4.3

During each iteration, Step 2 through 4 of Algorithm 1 are performed for a given row. Upon completion of an iteration, the sum of $\mathbf{X}$ is incremented by the calculated result, and then the next element of $document_{smallest}$ is read. Upon iterating through the full document, the value of $\mathbf{X}$ is made available to the CPU to perform the final calculations, as per Equation 2.5.

Unlike the case of the Pair-Wise approach, the treatment of Step 1 is affected by the unbounded nature of the One-to-N approach. If N is greater than the bit-width of the largest primitive data type that CUDA supports, which at the time of this thesis is a 64 bit value, then N must be less than or equal to 64. Therefore, a different

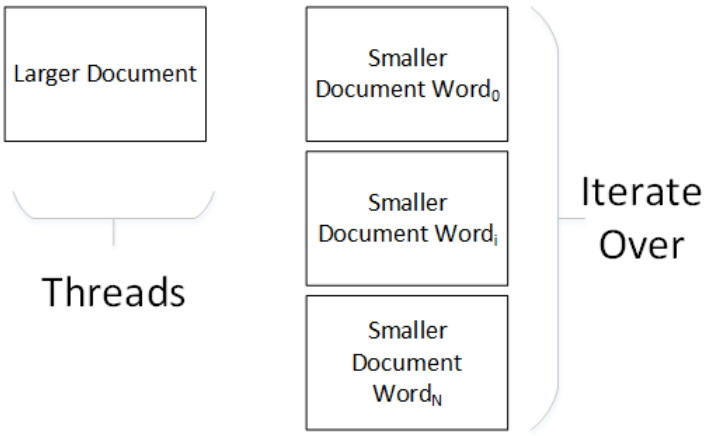

Figure 4.3: GPGPU One-to-N Approach Process for Performing Document Relatedness Calculations

approach than the flagging of matching words in the documents must be applied.

By incorporating Step 1 into the CUDA function to calculate the relatedness, this scalability issue can be resolved. Prior to performing the iterative processing for Steps 2 to 4, the documents must be checked for matching values. The threads will still read the elements of the $document_{largest}$, but will subsequently check each element of $document_{smallest}$ against the contents of the threads. If the elements match, then a common count is incremented, and the thread marks itself as a match. Once all the elements of $document_{smallest}$ have been processed, the threads then re-iterate over the next element of $document_{smallest}$ to perform Steps 2 through 4.

This alters the previous work flow of Steps 2 to 4, to allow the threads that are marked as a match to exclude themselves from further computation. The only exception to this is if the threads read an element from $document_{smallest}$ that matches a marked thread, meaning the processing of this row should be excluded from the calculation, the thread will force the pool to skip this iteration of the loop.

This results in a One-to-N approach that requires one CUDA kernel call to calculate the relatedness between any single document and N others, but like the Pair-Wise approach still requires that the results (number of matches, and the relatedness) are copied back to the host to perform Equation 2.5 for each of the N pairs. This results in an algorithm with the following performance:

Let $\mathbf{N}$ be $|document_{smallest}|$ and let $\mathbf{M}$ be $|document_{largest}|$. Due to the organizing

of the CUDA kernel, the elements of $document_{largest}$ are read a single time into memory followed by the reading of the elements of $document_{smallest}$ twice. The first read is intended to determine matches, and the second is for the relatedness calculation. This results in a total of $\mathbf{O}(2N + M)$ global memory accesses.

4.2 Modifications to the GPGPU Text Relatedness Algorithm to Address Variable Length Documents

While the previous two approaches provide a strong basis for estimating the GPGPU's ability to perform document relatedness, they fail to provide a system that can deal with documents of any length, specifically those larger than 1024 unique words.

The relevant limitation of CUDA hardware is that the maximum number of threads that a CUDA GPGPU currently supports is 1024, in a one dimensional grid. Therefore, when the two approaches are provided documents to calculate their relatedness, they will only map the number of threads to $|document_{largest}|$ in cases where $|document_{largest}| \leq 1024$.

To address cases where $|document_{largest}| > 1024$, both algorithms require alterations. Regardless of the approach, to work with variable length documents the CUDA kernels are required to be aware of $|document_{largest}|$. If $|document_{largest}| > 1024$, then the threads will have to read additional elements of $document_{largest}$.

Algorithm 4 illustrates how this additional reading is performed, taking advantage of the CUDA keywords that provide information on the *threadId*, and the number of threads per block ($blockDim.x$) This alteration indicates that the threads will no

Algorithm 4 Pseudo Code for CUDA Kernels to handle Variable Length Documents

```
1: threadId ← threadId.x
2: blockId ← blockId.x
3: if blockId ≤ |doc_1| then
4:     while threadId ≤ |doc_2| do
5:         /* Do kernel work */
6:         threadId ← threadId + blockDim.x
7:     end while
8: end if
```

longer be able to hold all unique values in memory in either of the approaches (Pair-Wise or One-to-N), as an individual thread is no longer responsible for retaining a single word of the document. This changes the algorithmic run-times and structure as described in the following sections.

4.2.1 Impact on the Pair-Wise Algorithm

The modifications to the Pair-Wise approach, as described in Section 4.1.1, require two additional CUDA kernel calls to calculate the relatedness between any two documents as shown below:

1. Find matching words between the two documents
2. Perform a reduction on the count of matches
3. Find the mean between the rows
4. Find the standard deviation between the rows
5. Find the relatedness of the rows
6. Perform a reduction on the relatedness of all rows

This result is then copied back to the CPU, which performs Equation 2.5, similar to Section 4.1.1.

As the Pair-Wise approach organizes the CUDA kernel calls by the number of blocks, and the number of threads per block, there are still $\mathbf{O}(NM)$ global memory accesses in the determination of the number of matches, and all the calculation steps aside from the final reduction which requires $\mathbf{O}(N)$ global memory accesses. This results in a bound of $\mathbf{O}(4NM)$ global memory accesses.

4.2.2 Impact on One-to-N Algorithm

Modifications to the One-to-N approach to support variable length documents include three additional CUDA kernel calls to calculate the relatedness between any two documents as shown below:

1. Find matching words between the sixty-four (64) documents

2. Find the mean between the documents

3. Find the standard deviation between the documents

4. Find the relatedness of the documents

This result is then copied back to the CPU, which performs Equation 2.5.

Most important of the alterations to the One-to-N approach for variable length documents is that N $\leq$ 64, which differs from its previously unbounded state. This bounding is required as the $document_{largest}$ is no longer able to stay in memory and thus requires that words of each document must be flagged and excluded from computation.

Additionally, upon completion of the calculations required for the mean, the whole document would have to be re-read for additional calculations, such as standard deviation and the document relatedness. The resulting algorithm requires $\mathbf{O}(N + M)$ global memory access for each kernel resulting in a bound of $\mathbf{O}(4N + 4M)$ global memory accesses.

4.3 Contrasting the GPGPU Approaches

Based on the analysis of the two presented approaches, their defining difference is the volume of memory accesses. Both approaches made heavy use of CUDA's thread and shared memory architecture to have each thread represent a single word pair. While memory efficient, this approach was hindered by the limited number of threads. Conversely, the refinements that enable the approaches to handle longer documents mean that a single thread cannot represent the data for a single word pair, and thus requires more accesses to global memory to accomplish the same amount of work.

4.4 Data Structures for Word Relatedness

Based on the decisions outlined in Section 3.5 to represent the WRM in the most compact form possible, easily indexing into this array for a given ($word_1$,$word_2$) pair is no longer possible. This section discusses different look-up strategies that are used by our proposed GPGPU implementations of the GTM.

Word 1	Word 1	Word 1	Word 2		
Word 1	Word 2	Word 10	Word 3		
1	.05	0.6	0.3		

Figure 4.4: Native format of WRM Data Structure.

This thesis presents four methods for word relatedness look-ups on a GPGPU. The first three are based on linear and binary search methods and are described in Section 4.4.1. The final method is based on perfect hashing and described in Section 4.4.2.

4.4.1 Search Strategies using a Sorted WRM

The WRM is composed of data elements organized so that each pair of ($word_1$, $word_2$) contains the relatedness between the words, as seen in Figure 4.4. By sorting the WRM in ascending lexicographical order and saving the resulting WRM back to storage, it is available for use as sorted data structure. Using the sorted WRM, the resulting look-ups to find relatedness between words from the documents can be performed via binary search, rather than linear probing. While this technique is an improvement over $\mathbf{O}(|WRM|)$ (linear search), the look up time can still be improved over $\mathbf{O}(\log(|WRM|))$ (binary search).

With the sorted WRM, one can further reduce the space required by remapping the WRM to an Index Array and a Value Array. This resulting data structure is illustrated in Figure 4.5, and is similar in concept to the Parallel Blocking Array data structure presented in [15].

The upper data structure from Figure 4.5 allows for the algorithm to search for a given $word_i$. If $word_i$ exists in the WRM, the search has the range of all possible ($word_i$, $word_N$) pairs via the last two indexes of the data structure. As this data structure is built from the sorted WRM, the data structure itself is sorted by $word_i$.

The lower data structure from Figure 4.5 provides both the value of $word_j$ for a

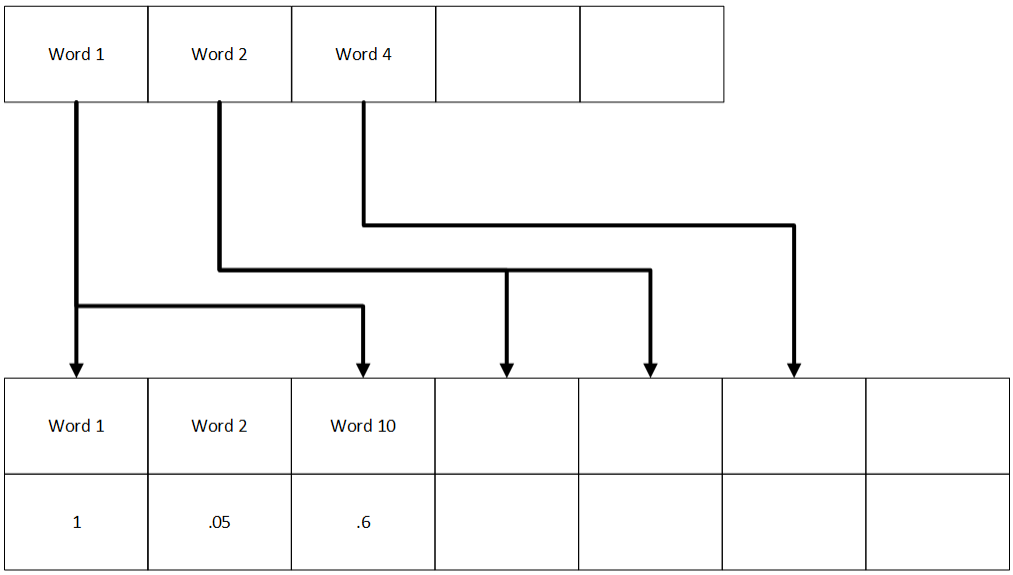

Figure 4.5: The WRM Remapped into an Index Array and the Corresponding Value Array.

given (word$_i$, word$_j$) pair and the corresponding relatedness. This data structure is a reduced version of the sorted WRM, omitting the word$_i$ value from the native format of that structure, seen in Figure 4.4.

By using this new data structure, the resulting look-ups into the WRM to find relatedness between words in the documents being compared can be performed via linear search, and by a double binary search.

The linear search allows for the searching of the initial data structure to find the word$_1$ indexes, prior to searching for the word$_2$ value and the word pair's relatedness between said indices in the secondary structure. Similarly, the double binary search allows for the binary searching of the initial look-up structure, to determine where in the secondary data structure the binary search should be performed.

Leveraging these techniques hopefully results in an improved performance over the standard binary and linear searches, as subsets of the WRM are searched rather than the WRM as a whole. This is captured as follows:

Let $\mathbf{N} = |word_i|$ values, and let $\mathbf{M} = |(word_i, word_j)|$ pairs. This results in:

1. $\mathbf{O}(N)$+$\mathbf{O}(M)$ memory access for linear search, and
2. $\mathbf{O}(\log(N))$+$\mathbf{O}(\log(M))$ memory access for binary search.

These methods will be evaluated in Section 4.4.3.

4.4.2 Search Strategies based on Hashing

The final approach to look-up strategies is the use of hashing to achieve look-ups in $\mathbf{O}(1)$ memory accesses. Based on the construction of the WRM, the look-up pairs (word_i, word_j) are ensured to be unique. Therefore, perfect hashing, a type of hashing which maps keys to values without collisions, can be used to access the WRM.

This work makes use of a perfect hashing library provided by [2], with slight modifications, including:

1. Modifying the library to read key values from a file
2. Modifying the library to handle the ASCII control characters

The alterations only required modification of the library's determination of string lengths, as it made use of Standard C's *strlen*() to determine the length of the keys. As shown in Algorithm 5, the keys created for use in this algorithm were made with a fixed length of eight ASCII characters.

Given the list of keys created from the two *WordID* pairs from each element in the WRM, the library returns a hash function. This hash function provides a unique mapping application from the keys to a value. This value is then used to re-build the WRM, where the new WRM is an array of **M** length, where **M** is defined such that $\mathbf{M} = 2^i \leq |WRM| \leq 2^{i+1}$ as per the library.

In this new WRM, only the indexes returned by the hash function would contain a word relatedness value. This value corresponds to the value held by the (word_1, word_2) pair in the initial WRM.

4.4.3 Evaluation of the Data Structures on WRM Retrieval

To determine the best data structure for use in the GPGPU versions of the algorithm, the sorted and hashing data structures were evaluated.

The evaluation to determine the performance of the data structures takes a two-phased approach. The first phase evaluates the performance of the data structures based on the wall-clock time to perform 242,427,791 word look-ups in the WRM. Based on the time required, the fastest performing data structure will then be further evaluated. If there are outliers in this first phase, they are independently explored further.

Algorithm 5 Creation of a perfect hashing key

$key \leftarrow std :: vector()$

if $\text{word}_1 \leq \text{word}_2$ **then**

$key_0 \leftarrow (word_1 \gg 24)\ \&\ 0xFF$

$key_1 \leftarrow (word_1 \gg 16)\ \&\ 0xFF$

$key_2 \leftarrow (word_1 \gg 8)\ \&\ 0xFF$

$key_3 \leftarrow (word_1)\ \&\ 0xFF$

$key_4 \leftarrow (word_2 \gg 24)\ \&\ 0xFF$

$key_5 \leftarrow (word_2 \gg 16)\ \&\ 0xFF$

$key_6 \leftarrow (word_2 \gg 8)\ \&\ 0xFF$

$key_7 \leftarrow (word_2)\ \&\ 0xFF$

else

$key_0 \leftarrow (word_2 \gg 24)\ \&\ 0xFF$

$key_1 \leftarrow (word_2 \gg 16)\ \&\ 0xFF$

$key_2 \leftarrow (word_2 \gg 8)\ \&\ 0xFF$

$key_3 \leftarrow (word_2)\ \&\ 0xFF$

$key_4 \leftarrow (word_1 \gg 24)\ \&\ 0xFF$

$key_5 \leftarrow (word_1 \gg 16)\ \&\ 0xFF$

$key_6 \leftarrow (word_1 \gg 8)\ \&\ 0xFF$

$key_7 \leftarrow (word_1)\ \&\ 0xFF$

end if

The second phase determines the performance of the selected data structure in terms of look-ups per second performed on the WRM. This calculation will be based on the time taken to process an average of 593,529,627,312 word pair look-ups in the WRM. This result is produced from the average performance of several runs of varying lengths from 133,000,000,000 to 1,100,000,000,000 word pairs.

The GPGPU architecture used for the evaluation of the performance of the WRM retrieval is a Nvidia GeForce 660 GTX, with 2 GB of RAM and 960 CUDA Cores, each clocked at 980 MHz, hosted in a PC running Windows 7 Home Edition. This GPGPU was used to run the executable produced using NVCC, Nvidia's CUDA compiler for CUDA 5.5. This test excludes the time needed to load the data structures and word pairs onto the GPGPU.

The results of the first phase are shown in Table 4.1.

Search Approach	Wall-Clock Time in Seconds	Space Required in Bytes
Linear Search	6082.48	854,118,732
Binary Search	5.68	1,135,497,760
Double Binary Search	6.35	854,118,732
Perfect Hashing	1.99	1,107,296,499

Table 4.1: Wall Clock Time Taken to Perform 242,427,791 WRM Queries on a GPGPU for a given Data Structure

The results presented in Table 4.1 show that the fastest data structure is the perfect hashing technique, as it performs the same number of look-ups as the next closest performing structure in nearly a third of the time. While the cost in terms of storage is nearly 50% of the targeted GPGPU memory, the search strategy delivers a significant performance increase over the both the standard binary and double binary search data structures, which consume either the same amount of memory or a $\frac{1}{3}$ less memory.

Using the perfect hashing search technique, the second phase of the evaluation varies the volume of the word pair look-ups that are used in this evaluation, resulting in the average number of word pairs cited in Table 4.2. Each of these word pair lists were evaluated, and the results were used to produce the reported performance of perfect hashing.

Average Number of Word Pairs	Average Number of Look-Ups per Second
593,529,627,312	377,989,297

Table 4.2: Word Similarity Look-Ups per Second on the GPGPU for the Perfect Hashing Data Structure

4.5 Document Loading

Since GPGPUs possess a limited amount of memory compared to a workstation, server, or desktop computer, it is assumed that there will be corpora that will not fit entirely in the GPGPU's memory along with the data structures required to perform document relatedness. To effectively process these corpora, any proposed GPGPU solutions must be able to copy elements of a given corpus from the CPU to the GPGPU in an effective manner.

Whenever possible, the loading approaches used will take advantage of an ability Nvidia's hardware supports, called streaming. Streaming allows a programmer to specify a series of operations to be executed sequentially. This functionality ensures that a specific series of instructions, such as memory copies and kernel executions, is ensured to be completed prior to the execution of the next sequence of events in the stream. This allows for the avoidance of dirty, or conflicted, reads. This translates well to the proposed Pair-Wise approach for document relatedness, as each Pair-Wise calculation is a series of sequential events that do not affect any other pairs of documents being compared.

In this thesis, we have proposed two distinct approaches for performing N:N document relatedness. The Pair-Wise approach, which computes N:N document relatedness as a series of 1:1 calculations, and the One-To-N approach, which produces N:N document relatedness as a series of 1:N document relatedness evaluations. Based on these approaches, a base document, the 1 in our 1:1 or 1:N, is fetched from the CPU and held on the GPGPU until the base document has been compared with all N documents. At this point a new base document can be fetched from the CPU.

The following sections deal with the loading schemas that were investigated to provide the documents to compare our base document with.

4.5.1 Singleton

The singleton approach provides both a baseline method for loading documents from the CPU to the GPGPU, and the only loading mechanism possible for the One-to-N approach to use due to the control structures required for that approach. Thus the evaluations of the loading approaches will be explained from the point of view of the Pair-Wise Approach.

The singleton approach performs a single copy of one or more comparison documents from the CPU to the GPGPU. Upon completion of the relatedness calculations between the base document and the comparison document, a new document is retrieved from the CPU and relatedness calculations are repeated. This process is repeated until the base document has been compared with the rest of the corpus. Once this is completed, a new base document is loaded from the CPU, and the process is repeated for the next document until the N:N relatedness has been calculated.

4.5.2 Stride

Similar to the singleton document loading method, the stride approach allows for the base document to be loaded into GPGPU memory and retained until the 1:N document relatedness calculations are completed. Unlike the singleton method, each comparison document of the stride is loaded from the CPU to the GPGPU in a stream, with the document relatedness calculations for each $stride^{th}$ document against the base document queued. Figure 4.6 illustrates this process.

Due to CUDA's architectural limits, the maximum width of a stride is thirty-two (32) documents, as that is the maximum number of streams the hardware currently supports. Upon completion of all comparisons within a given stride, the next stride is retrieved from the CPU and relatedness calculations are performed. This process is repeated until the 1:N document relatedness is completed, at which point a new base document is selected and the process repeats.

The width of a stride is determined based on the present location of the algorithm in the 1:N relatedness calculation. For instance, if there are fewer than 32 documents remaining, then the stride is clamped to that value, otherwise the maximum stride value is taken.

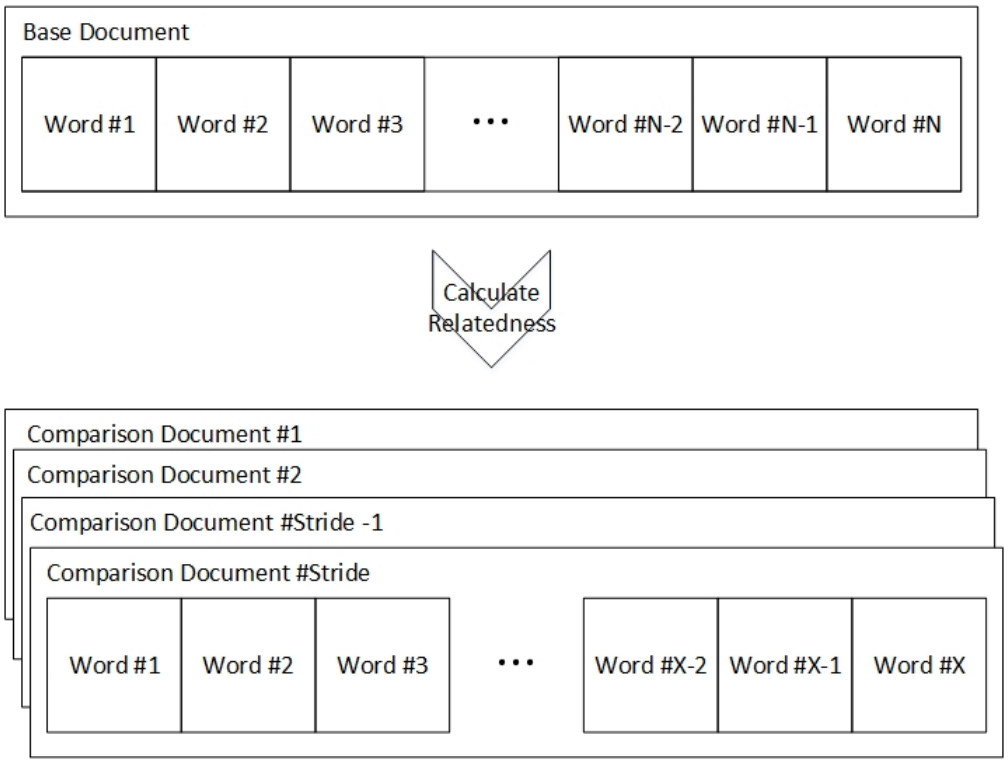

Figure 4.6: Display of Pair-Wise Approach When Stride Loading is Used

4.5.3 Grid

The grid approach is similar to the stride loading approach, however, rather than a linear 1:N approach to computing the N:N document relatedness matrix, the grid loading approach alters how the documents are held in GPGPU memory. This allows for the calculation of a X:N documented relatedness approach, where X is the number of base documents held for a single iteration through the Grid. Similar to the other methods, each comparison document is loaded into GPGPU memory, and compared against a base document, as Figure 4.7 illustrates.

The document comparisons are placed into CUDA streams, between the documents that comprise the row (base) documents and those that compose the columns, explained in Algorithm 6.

4.5.4 All-in-Memory

The final approach to be considered applies only to cases where the corpus will fit into GPGPU memory in its entirety. In this approach, the corpus is loaded into memory,

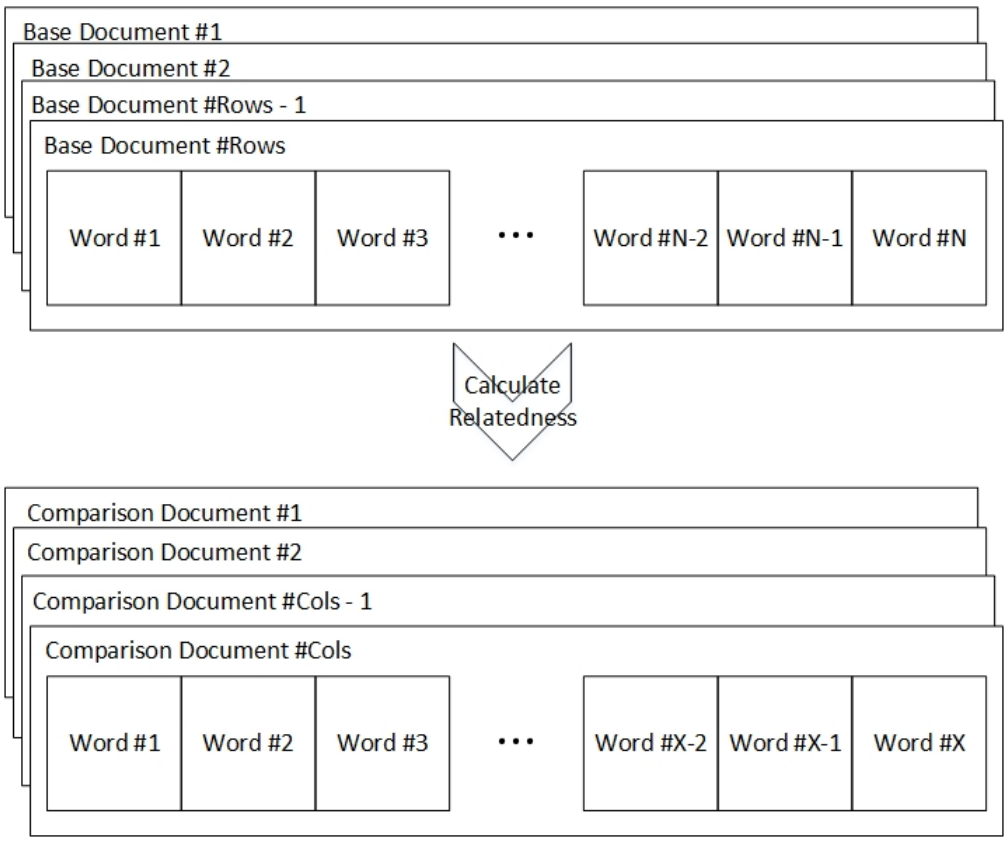

Figure 4.7: Display of Pair-Wise Approach When Grid Loading is Used

Algorithm 6 High Level Description Grid Approach

1: **for** $i \leftarrow 0; i < N; i \leftarrow i + Grid_{Rows}$ **do**
2: **for** $k \leftarrow 0; k < Grid_{Rows}; k++$ **do**
3: $baseDocument_k \leftarrow Corpus_{i+k}$
4: **end for**
5: **for** $j \leftarrow i; j < N; j \leftarrow j + Grid_{Cols}$ **do**
6: **for** $k \leftarrow 0; k < Grid_{Cols}; k++$ **do**
7: $compareDocument_k \leftarrow Corpus_{j+k}$
8: **end for**
9: **for** $x \leftarrow 0; x < Grid_{Rows}; x++$ **do**
10: **for** $y \leftarrow 0; y < Grid_{Cols}; y++$ **do**
11: Place the following in a CUDA Stream
12: CALCULATE RELATEDNESS($baseDocument_x, compareDocument_y$)
13: **end for**
14: **end for**
15: **end for**
16: **end for**

then a base document is selected. The initial base document is then compared against the remaining documents.

The comparison documents are treated as a stride, with the length of the maximum number of concurrent kernels that the CUDA hardware supports and appropriate bounds-checking. Each of the documents that make up the stride are compared against the base document in parallel. This process is repeated until N documents have been compared, at which point a new base document is selected.

4.5.5 Performance of the Document Loading Methods Strategies

To determine the most appropriate loading method, an experiment using the Pair-Wise document relatedness approach was devised. This experiment involved using the ACM Dalhousie Abstract corpus' first two thousand (2000) documents to determine how document loading would affect the time required to calculate their relatedness. Aside from modifying the applied document loading strategy, each version of the experiment made use of the same elements of the corpus, and used the same WRM search strategy. The results of this experiment can be seen in Table 4.3.

Each part of the experiment ran the executables that were produced using NVCC, Nvidia's CUDA compiler for CUDA 5.5 and made use of OpenMP 2.0 for concurrent kernel launches. The GPGPU used for these experiments was a Nvidia GeForce 660 GTX, with 2 GB of RAM and 960 CUDA Cores, each clocked at 980 MHz, hosted in PC running Windows 7 Home Edition.

Loading Approach	Documents Processed per second
Singleton	7,711
Stride	9,002
Grid	2,106
All-in-Memory	10,081

Table 4.3: Impact of Document Loading Approach on Document Relatedness

The results of the performance of the grid approach merit some discussion. While this approach should theoretically be the most efficient method of performing document relatedness, the observed performance does not support this. Based on the copying of the comparison documents in streams, the other comparison documents are not guaranteed to be in GPGPU memory until the stream responsible for the copy

is performing the document relatedness between a document pair. In order to ensure that documents are available, the GPGPU must either block until each stream's first document relatedness calculations are ready to begin, or copy the comparison documents in a singleton load. Performing either approach did not improve the results, as the GPGPU accesses the global memory more frequently for the comparison documents and base documents due to the loss of locality of reference and cache misses when compared to the other methods.

The results of this experiment show that if all documents can fit into GPGPU memory, then the All-in-Memory approach should be used. While this will work for a smaller corpus such as the ACM Dalhousie Abstract collection, it is not applicable for larger corpora, such the Gutenberg Dataset. In any case where the corpus is too large for an All-in-Memory approach, the stride approach is the most efficient for document similarity computations.

4.6 The Optimized GPGPU Approaches

Based on the evaluations and design considerations presented in this chapter, two approaches for using GPGPUs to calculate document relatedness are recommended: the Pair-Wise approach and the One-to-N approach. This thesis applies these two approaches to calculate relatedness between varied length documents, rather than documents assumed small enough to be held in thread memory.

Each of these approaches will make use of perfect hashing as the WRM retrieval method based on the performance observed in Section 4.4.3. For document loading, the One-to-N approach will use a singleton load of N, and the Pair-Wise approach will use the stride method for document loading.

The proposed framework is expected to deal with corpora of any size, so the stride method will be further refined so that the stride length will be tunable based on the current size of the documents being compared against the base document. This tuning can be summarized as per Algorithm 7.

With all relevant considerations and methods defined, the GPGPU approaches are now ready for evaluation against a baseline and against each other.

Algorithm 7 Tuning the Stride Loading Amount

Require: The corpus to be loaded into CPU memory, and the footprint of the GPGPU WRM Data Structure to be known.

Ensure: The corpus fits in the CPU memory

1: **for** $i \leftarrow 0; i < |Corpus|\,; i++$ **do**
2: **for** $j \leftarrow i; j < |Corpus|\,;$ **do**
3: $stride \leftarrow STRIDEMAX$ ▷ Either 32 or 64 depending on approach
4: **if** $stride > |Corpus| - j$ **then**
5: $stride \leftarrow |Corpus| - j$
6: **end if**
7: $arrayOfLenghts \leftarrow$ findLengthsOfDocuments($stride$)
8: $largestDocument \leftarrow$ greatestLength($arrayOfLenghts$)
9: $stride \leftarrow$ calculateSpaceRequired($largestDocument$)
10: **if** $stride = 0$ **then**
11: exit(1)
12: **end if**
13: **if** $stride\ |Corpus| - j$ **then**
14: $stride \leftarrow |Corpus| - j$
15: **end if**
16: Copy the *stride* documents to the GPGPU
17: Create the supporting variables to hold the *stride* documents
18: Perform document relatedness work
19: $j \leftarrow j + stride$
20: **end for**
21: **end for**

Chapter 5

Computing GTM Document Relatedness on a Multi-Core System

This chapter introduces two multi-core approaches for calculating document relatedness that will be used to evaluate the performance of the GPGPU approaches. Similar to Chapter 4, the work presented in this chapter will map the generalized algorithm for calculating document relatedness to the CPU. The data structures used to store and access the WRM are then investigated and evaluated.

5.1 Construction of a Multi-Core Algorithm and Mapping

To construct a multi-core performance benchmark, the generalized algorithm for document relatedness must be mapped to a multi-core CPU, hereafter referred to as a CPU. At a high level the main difference between the approach for the CPU and the high-level approach for the GPGPU, see Algorithm 2, is that the CPU approach does not require memory transfers, as the CPU is able to hold all required data in memory.

Much like the GPGPU version, the CPU benchmark will require an investigation into the data structures required to search and store the WRM. Unlike the GPGPU approaches, the CPU baseline can make use of the C++ Standard Template Library (STL). The STL provides a repeatable and common implementation with guaranteed performance for the templates [data structures] regardless of target architecture, and without requiring the data structures to be reproduced in project specific code.

Additionally, to make the CPU performance more comparable to a multi-threaded GPGPU, OpenMP will be used to speed up the computation. This is to address existing research [12] that contrasts GPGPU performance gains and has shown that without a proper basis of comparison, any reported speed-up is not accurate.

The parallelization provided by OpenMP is leveraged to speed up the operations performed in the loops that compose the algorithm described in Algorithm **??**. This

is accomplished via the ***#pragma parallel for*** command around the loop to parallelize. The CPU's approach to parallelization using OpenMP can be performed in two ways:

1. The process of comparing a given document$_i$ against document$_j$ can be parallelized, giving each thread a unique (document$_i$, document$_j$) pair to process.

2. The process of comparing a given document$_i$ against document$_j$ can be parallelized, where each thread works on the same (document$_i$, document$_j$) pair in parallel.

These two approaches, discussed in the following sections, each make use of common code as OpenMP provides the **#pragma** commands to handle either approach without requiring code changes from a single threaded approach.

5.1.1 Parallelizing for Document Comparison Throughput for the Baseline

The parallelization of this approach is captured in Algorithm 8. As this algorithm shows, the modifications required to parallelize for document pair throughput are trivial.

5.1.2 Parallelizing for an Individual Document Comparison Approach for the Baseline

The parallelization required for this approach is performed on the following functions defined in Algorithm **??**: RemoveMatches() and CalculateRelatedness(). The modifications required to support the parallelization are captured in Algorithms 9 and 10. This approach uses the best performing data structure as determined in Section 5.2 for the WRM look-up.

5.2 Determining the Data Structures Used for WRM Retrieval

To determine the best data structure to hold WRM, the following options were proposed:

Algorithm 8 CPU Benchmark - Parallelizing For Document Throughput

Require: The corpus and WRM are pre-processed. The user provides a start and stop point for the documents to be compared

Ensure: The files exist

```
wrm ← READ WRM
corpus ← READ CORPUS(start,stop)
for i ← 0; i < |Corpus| ; i++ do
    baseDocument ← Corpus_i
    # pragma parallel for
    for j ← i; j < |Corpus| ; j++ do
        compareDocument ← Corpus_j
        doc_1, doc_2 ← REMOVE MATCHES(baseDocument, compareDocument)
        relatedness ← CALCULATE RELATEDNESS(doc_1, doc_2, wrm)
    end for
end for
```

1. STL Map. This is a sorted map that works on a (key,value) pair structure. In this baseline the key is the STL pair object that is composed of the ($wordId_i$ $wordId_j$) pairs from the WRM, and the value is the similarity between these two words. Querying the STL Map is a function of $\mathbf{O}(\log(N))$.

2. Binary Search. This is identical to the WRM structure of the same name proposed in Section 4.4.1.

3. Double Binary Search. This is identical to the WRM structure of the same name proposed in Section 4.4.1.

4. Perfect Hashing. This is identical to the WRM structure of the same name proposed in Section 4.4.2.

5. STL UnorderedMap. Similar to the STL Map, this data structure works on a (key, value) pair system; however unlike the STL Map, the UnorderedMap performs a hash on the values and places them into buckets.

Algorithm 9 CPU Benchmark - High-Level Parallelizing For Individual Document Comparison

Require: $|Doc_1| \leq |Doc_2|$

function CALCULATE RELATEDNESS(Doc_1, Doc_2)

Step One: For each word of Doc_1: if it is not flagged, proceed to Step Two. If it is flagged, advance to the next word of Doc_1

Step Two: In a **#pragma parallel for**, find the WRM(Doc_{1i}, Doc_{2j}) if Doc_{2j} is not flagged.

Step Three: In a **#pragma parallel for**, find the mean of the WRM values retrieved in the previous step.

Step Four: In a **#pragma parallel for**, find the deviation of the WRM values retrieved in **Step Two**.

Step Five: Find the standard deviation using the value from the previous step.

Step Six: In a **#pragma parallel for**, find the values of WRM(Doc_1, Doc_{2j}) which are greater than sum of the mean and standard deviation. In the same loop, find the mean of those values.

Step Seven: Add this resulting mean to the relatedness value.

Step Eight: Upon processing of all of the words of Doc_1 return the relatedness value

end function

Algorithm 10 CPU Benchmark - Parallelizing For Individual Document Comparison

```
function REMOVE MATCHES(document_i, document_j)
    for i ← 0; i < |document_i| ; i++ do
        #pragma parallel for
        for j ← 0; j < |document_j| ; j++ do
            if Document_j[j] = Document_i[i] then
                Flag elements in both documents as match
            end if
        end for
    end for
return Doc_1, Doc_2
end function
```

Before discussing the evaluation of these data structures and their performance, it is important to provide context as to why the STL containers were selected. By selecting the STL Map, the binary search approaches can be contrasted against a known approach that performs in $\mathbf{O}(\log(N))$ time and works with any type of data. This provides a baseline for the specific binary search techniques developed and employed in this thesis.

Similarly, the STL UnorderedMap was selected to provide a relative comparison to the Perfect Hashing Library used. To that end, the STL algorithm made use of a predefined hash function to create entries for all of the values and keys passed to it. Unlike the Perfect Hashing Library, the UnorderedMap was evaluated through two different key generation approaches. The first key creation strategy was from Algorithm 5, save for the values being stored in an std::string vice an array. The second strategy is defined as per Algorithm 11. The approaches were selected to test whether the key data type would affect the results.

Algorithm 11 Alternate Approach For the Creation of an UnorderedMap key

$key \leftarrow 0$
if $\text{word}_1 \leq \text{word}_2$ **then**
    $key \leftarrow (word_1 \ll 32) \mid word_2$
else
    $key \leftarrow (word_2 \ll 32) \mid word_1$
end if

5.2.1 Evaluating the Data Structures for the WRM Retrieval in the Baseline

To evaluate the performance of the data structures, a single-phased approach is used. The data structures are evaluated through the wall-clock time taken to perform 242,427,791 word look-ups from the WRM. Based on the time taken, the data structure that performed the fastest will be selected. If there are outlying results, they will be explored further prior to determining the best data structure.

The hardware used for the evaluation of the data structure was CGM6, a Linux server composed of 2 Intel Xeon ES-2650 processors, each with 8 cores and Hyper

Threading enabled, with shared access to 264 GB of RAM. This server ran the experiment using executables that were produced using GCC 4.4.7 and that version of GCC's OpenMP.

Each of the resulting executables, one per data structure, were identically coded aside from the differing elements of the data structures used. Note that each of the reported times exclude the time needed to load the data structures and word pairs into memory.

Search Approach	Wall-Clock Time in Seconds	Space Required in Bytes
Binary Search	189.03	1,135,497,760
STL Map	23.51	5,544,000,000
Double Binary Search	11.32	854,118,732
STL UnorderedMap 1	11.28	3,960,000,000
STL UnorderedMap 2	5.48	3,960,000,000
Perfect Hashing	282,020	1,107,296,499

Table 5.1: Wall-Clock Time Taken to Perform 242,427,791 WRM Queries on the CPU for a given data structure using 64 OpenMP threads

The results of this evaluation are shown in Table 5.1. The presented results are in line with expectations. As expected, of the divide and conquer approaches, the Double Binary Search is the ideal candidate, and hashing is the fastest way to access the WRM. However, despite the excellent performance of the perfect hashing routine observed on the GPGPU (see Section 4.4.3) this performance does not exist on the CPU.

Based on the results of the STL UnorderedMap, key construction can play a role in the time taken to process an index into the WRM. However, when the STL UnorderedMap made use of the same key forging algorithm as perfect hashing, Algorithm 5, it completed the processing of the word pairs in a significantly shorter period of time. Since the key construction technique does not appear to be the problem, and the hashing approach is not a performance impediment, one can conclude that the issue with perfect hashing results is the library used.

In the 15 years since this hashing library was created, the instruction sets provided in modern CPUs have evolved to expect operations and values larger than 8-bit words, as evidenced by the rise of 64-bit instruction sets. This change in instruction

sets would require extensive shifting prior to being subjected to the bit manipulations required for the Perfect Hashing library to be effective. Similarly, this explains why the GPGPU did not experience noticeable issues with the hashing library as CUDA architecture requires several instruction sequences to complete a single 64-bit operation.

As the STL UnorderedMap, when creating a key using Algorithm 11, took the least amount of time to perform the look-ups, it was selected to be used as the data structure for the CPU benchmarks.

5.3 Recommended Baseline Approaches for Document Relatedness

The two proposed CPU benchmark approaches can be defined by the level of parallelization they leverage with respect to evaluation of the document relatedness space. The first method, introduced in Section 5.1.1, suggests coarse-grained parallelization where each document pair to be evaluated is assigned its own thread. The other method, proposed in Section 5.1.2, proposes fine-grained parallelization, where each document pair to be evaluated is worked on by all threads. Regardless of the approach, each of these potential benchmarks will access the WRM using the STL UnorderedMap, as this was proven to be the best suited data structure.

In Chapter 6, the CPU benchmarks will be evaluated against each other, and the best performing benchmark will be used to evaluate the proposed GPGPU approaches.

Chapter 6

Evaluation of the GPGPU Approach

This chapter evaluates the document relatedness approaches for the GPGPU and CPU as discussed in Chapters 4 and 5 respectively. We will begin by discussing the specifics of the evaluation methodology and how it will be applied to the selected approaches, followed by an analysis using the selected corpora, and finally a presentation of the results. Both CPU GTM approaches described in Chapter 5 are evaluated, and the one with the best performance will be selected as the benchmark against which the GPGPU approaches will be compared.

In Chapter 4, two GPGPU approaches were described. The first was performed in shared GPGPU memory but was only applicable for documents which contained less than 1024 unique words. The second method used a combination of shared and global GPGPU memory but was much more widely applicable, only requiring that the documents fit in the GPGPU's global memory. We first compared the GPGPU approaches using the Association for Computing Machinery (ACM) Dalhousie Abstract Collection corpus (which had documents that were less than 1024 words in length).

We then compared the global memory GPGPU method with the CPU benchmark using both the ACM Dalhousie Abstract Collection corpus and a 'real life' text corpus drawn from the Gutenberg Collection [11].

6.1 Configuration of the Experiments

The experiments make use of two different target architectures, the traditional Intel x86 architecture and Nvidia's GPGPU architecture. Due to the architectural differences discussed earlier, they cannot both run the same executable.

The CPU and GPGPU platforms used different operating system environments

and configurations. For the CPU version, experiments were performed with an executable produced using GCC 4.4.7 and that version of GCC's OpenMP. This executable was then evaluated on a Linux server composed of 2 Intel Xeon ES-2650 processors, each with 8 cores and Hyper Threading enabled, and shared access to 264 GB of RAM. The GPGPU experiments were performed using an executable produced using NVCC for CUDA 5.5, compiled for Compute 1.0 architecture. The resulting executables were then evaluated on a Windows 7 Home Edition PC comprised of a quad core Intel i5, clocked at 3.40 GHz and paired with a GeForce 660 GTX.

The timings reported for the experiments performed in this chapter were derived as follows:

1. All data structures and documents were pre-loaded into the host memory of the CPU.

2. Timings for the CPU methods include the time from the start of the document similarity calculations to the final document similarity result being written onto disk.

3. Timings of the GPGPU methods include all transfers of data between the CPU and GPGPU and all of the similarity computations. The timing also includes the time taken to transfer the results of the similarity computation from the GPGPU to the CPU, and to write the results to disk.

6.2 Evaluation Method

The evaluation of the proposed CPU and GPGPU approaches focuses on their ability to calculate document relatedness. Regardless of the specific hardware, any approach will be evaluated by measuring the document relatedness calculation rate that is observed when processing a corpus. The rate of calculation can be expressed in two possible rates:

1. The number of documents per second (DPS) that have been processed

2. The number of words per second (WPS) that have been processed

The measurement of DPS may initially seem the most relevant in assessing approaches to calculating document relatedness, however, it prevents clear comparisons between different corpora. Given that different corpora will rarely have individual documents of consistently uniform lengths, an analysis of only the DPS calculation could potentially mask the complexity of the results. WPS calculations allow for inter-corpora comparisons, and provide a method to determine the comparable work of computing document relatedness with documents of any length.

With these considerations taken into account, both approaches will generally be evaluated through WPS metrics, allowing for discussions regarding the throughput across corpora and on an individual corpus. DPS metrics are only considered when another level of analysis is required, and are only used in conjunction with a WPS metric.

The WPS is calculated with the total number of words that are compared between the relevant number of documents using the generalized algorithm outlined in Section 2.2. For example, the total number of words included in a Pair-Wise calculation would be the number of words compared between a pair of documents, where a 1:N calculation would include the total number of words compared between the base document and the series of N documents. The total number of words captures the number of elements that should be present in the matrix proposed in GTM Step Two of the generalized algorithm. This total is then divided by the amount of time, in seconds, taken to complete the word relatedness calculation to ultimately determine the number of words processed per second.

Equation 6.1 illustrates how the WPS is calculated for computing N:N document relatedness.

$$WPS = \sum_{i=0}^{N} \sum_{j=i}^{N} \frac{|D_i|\,|D_j|}{Time(i,j)} \tag{6.1}$$

where: N = The number of documents to find relatedness between
D = A document from the corpus
$T(i,j)$ = The time taken to calculate and record document relatedness

Equation 6.1 makes the assumption that the time taken to determine if there are matching words is negligible in the overall calculation, and the true measure is in

the algorithmic work conducted in GTM Steps Two through Five, as captured in Algorithm 1.

6.3 Data Sets Used in Experimentation

The following corpora were selected for use in evaluating the framework:

1. The ACM Dalhousie Abstract Collection, which is composed of 43,452 abstracts from papers published in the ACM digital library and prepared for use by Dalhousie students.

2. The Gutenberg Collection, provided by [11], includes a subset of the Project Gutenberg literary works. This subset provides a collection of 3,036 works of literature that are in the public domain.

For any document relatedness experiments, the content of the given corpus must be known in order to determine the effectiveness of the proposed approaches. The following section will discuss the content characteristics of the two selected corpora, including an outline of each of the corpus' main parameters, the average length of a given subset of the corpus, and the total number of words that must be processed in that subset to calculate document relatedness as per the generalized algorithm.

6.3.1 ACM Dalhousie Abstract Collection

The ACM Dalhousie Abstract Collection was processed according to Section 3.6, and the first 10,500 documents were used for the relatedness calculations. Figure 6.1 shows the average length per processed document for the given subset of the ACM Dalhousie Abstract corpus, as well as the standard deviation for the processed document length. Figure 6.2 illustrates that the growth rate of the number of words to compare appears to grow in a quadratic fashion as the subset incorporates more documents from the corpus.

6.3.2 Gutenberg Collection

The Gutenberg Collection was also processed according to Section 3.6, and the first 200 documents were used for relatedness calculations. Additionally, these documents

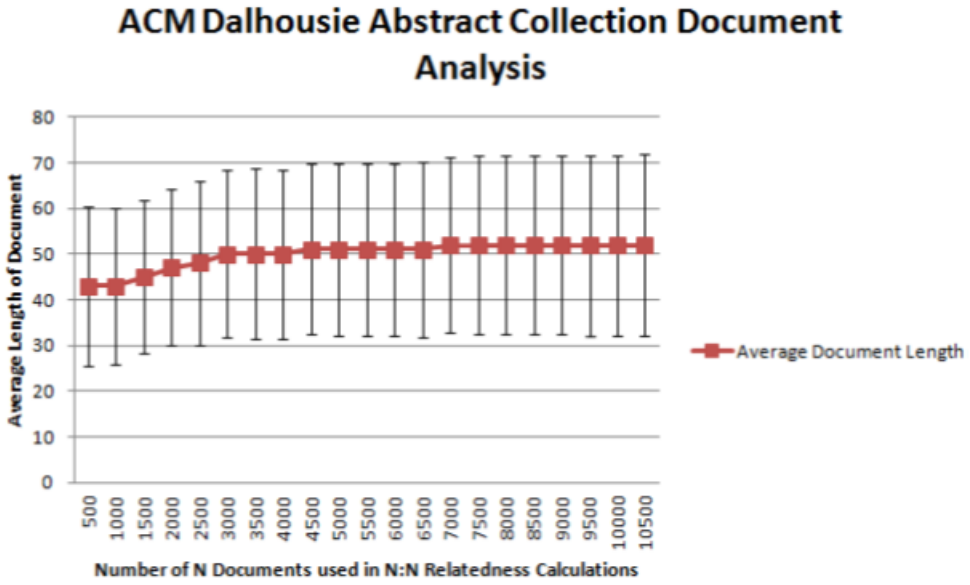

Figure 6.1: The Average Document Length and Deviation for a Given Segment of the ACM Dalhousie Abstract Collection

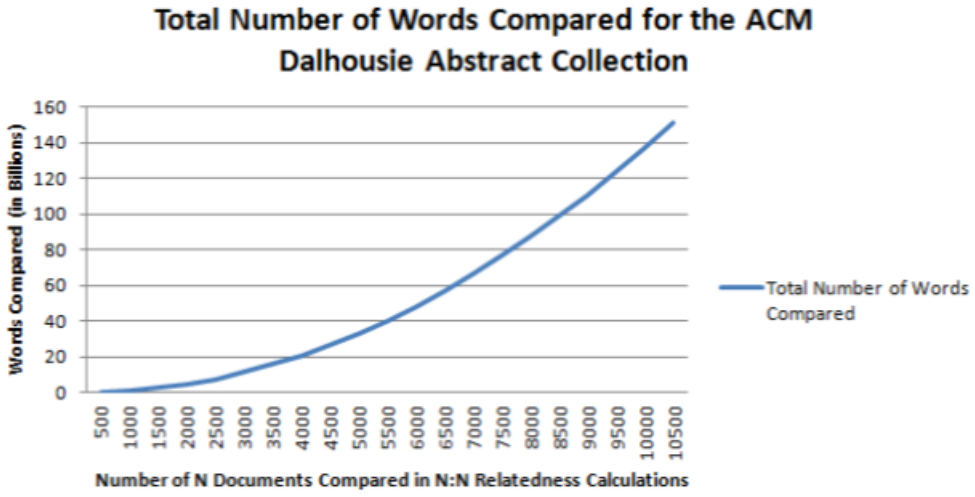

Figure 6.2: The Number of Words Required to be Compared to Complete the Document Relatedness for a Given Segment of the ACM Dalhousie Abstract Collection

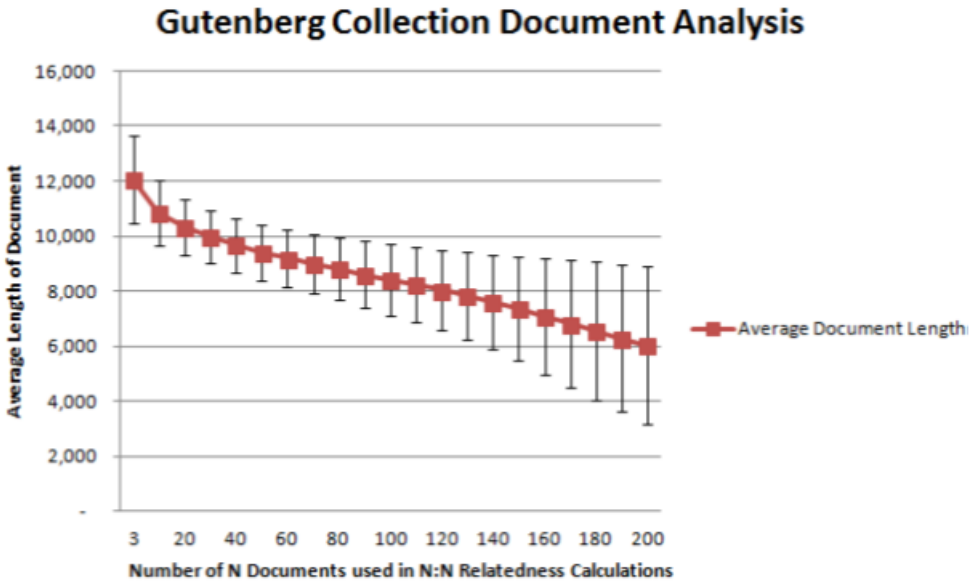

Figure 6.3: The Average Document Length and Deviation for a Given Segment of the Gutenberg Collection

were sorted by their length, such that the larger documents of the selected subsets were evaluated first. Figure 6.3 shows the average processed document length for the given subset of the Gutenberg Collection corpus, as well as the standard deviation in the processed document length. Figure 6.4 illustrates the growth rate of the number of words to compare as the subset incorporates more documents from the corpus.

6.4 Determining the CPU Benchmark Performance

In order to draw relevant conclusions for the effectiveness of a GPGPU implementation of the GTM, the GPGPU approach in question must be compared against an efficient CPU implementation [12]. This thesis presented two possible CPU approaches in Section 5.1: one focusing on parallelizing a document pair as thoroughly as possible, and another focusing on parallelizing the throughput of document pairs.

In order to determine which CPU approach will act as the benchmark against which we evaluate the GPGPU approaches, the two CPU approaches were evaluated in separate experiments to determine which generates the best rate of WPS. The approach with the highest overall WPS becomes the baseline approach for comparison. These experiments serve to evaluate the CPU approaches on various subsets of the

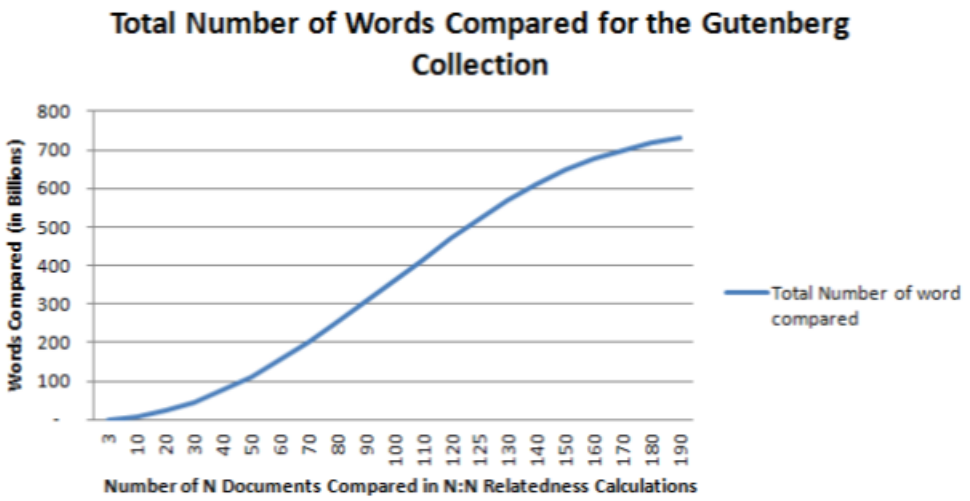

Figure 6.4: The Number of Words Required to be Compared to Complete the Document Relatedness for a Given Segment of the Gutenberg Collection

first 1000 documents of the ACM Dalhousie Abstract Collection, and the first 100 documents of the Gutenberg Collection.

Figure 6.5 illustrates the performance of the CPU Throughput approach on the ACM Dalhousie Abstract Collection. It shows the number of words compared per second as a function of the number of documents compared. We observe that as the number of documents to be compared increases, the amount of work required and the computed WPS increases. The WPS rate increases until around 400^2 documents are compared for their relatedness, after which the rate of improvement begins to slow. Note that this figure illustrates that about 50 million WPS can be processed using this CPU approach.

Figure 6.5 also illustrates the performance of the CPU Parallelized Individual approach on the ACM Dalhousie Abstract Collection. We observe that when compared to the CPU Throughput Approach, this method shows significantly lower performance. The CPU Parallelized Individual approach achieves about 33,000 WPS versus the 50 million WPS achieved by the throughput method. This is presumably because there are not enough words present in the documents of ACM Dalhousie Abstract Collection to efficiently parallelize a 1:1 document relatedness computation on a 16-core processor. The overhead of using OpenMP in this case simply overcomes the anticipated and expected performance gains.

Figure 6.6 illustrates the performance of the CPU Throughput approach on the

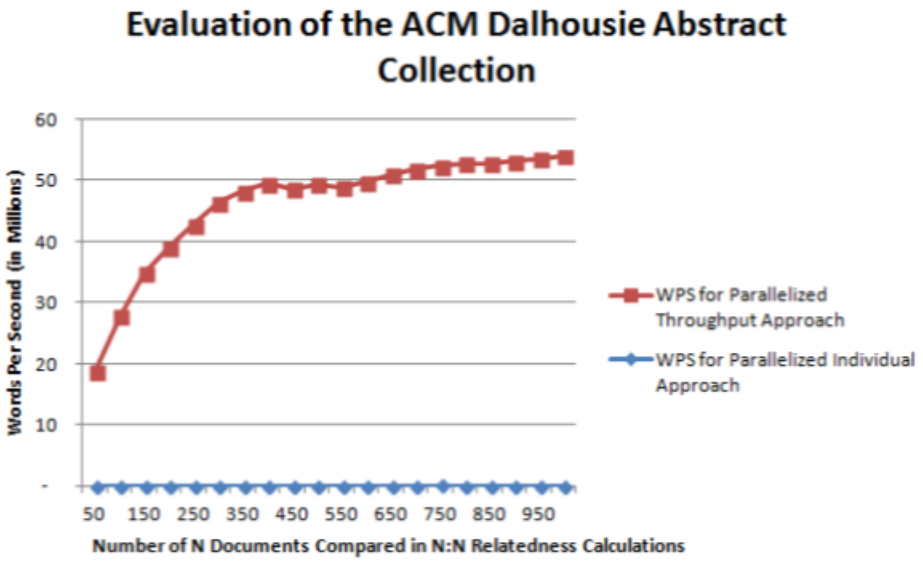

Figure 6.5: The Observed Performance of the CPU Throughput Approach and CPU Parallelized Individual Approach When Processing Segments of the ACM Dalhousie Abstract Collection

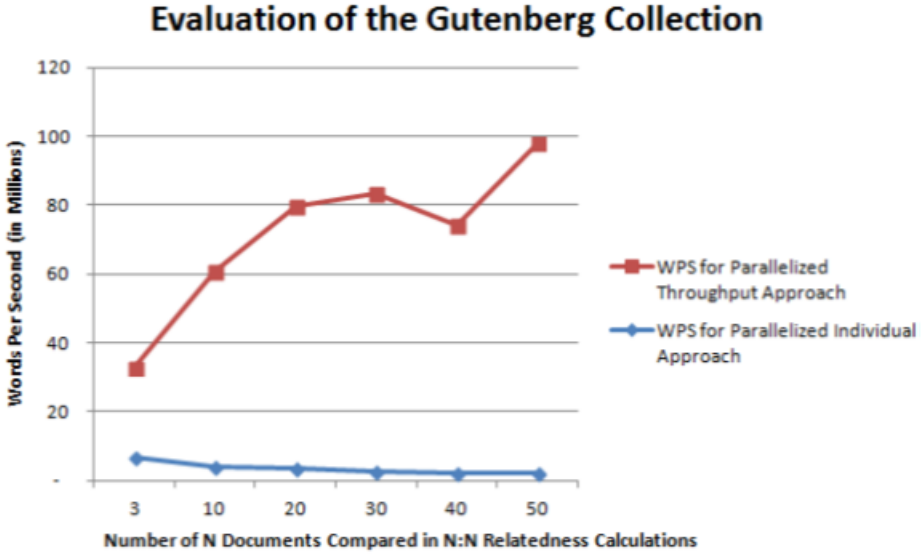

Figure 6.6: The Observed Performance of the CPU Throughput Approach and CPU Parallelized Individual Approach When Processing Segments of the Gutenberg Collection

Gutenberg Collection. It shows the words compared per second as a function of the number of documents compared. As the number of documents compared increases, the rate of WPS increases as well. We observe that performance drops off around 40^2 documents before returning to a positive slope. The decrease is better understood when we examine Figure 6.3. This figure demonstrates that at 40^2 documents, the average document length of the corpus begins to increase at a greater rate than its maximum of approximately 6,000 words per document. The rate of WPS increases from the point of 20^2 documents compared until the end of the experiment. Note that this figure illustrates that about 100 million WPS can be processed using this CPU approach.

Figure 6.6 also illustrates the performance of the CPU Parallelized Individual approach on the Gutenberg Collection. We observe that the performance of this approach is significantly better than the performance with the ACM Dalhousie Abstract Collection. We see a continued decline in performance as the average document size of corpus decreases. This figure confirms that the CPU Parallelized Individual approach is very much dependent on the length of the document(s) in question. The performance observed in this figure shows that longer documents allow for the overhead of using OpenMP to be masked by the increase in performance. Unfortunately, while the CPU Parallelized Individual approach was aided by the longer documents, with a WPS rate of around 3 million, the CPU Throughput approach was observed to have a rate of WPS of 100 million.

Given the superior performance of the CPU Throughput approach, it will be used as the benchmark for evaluating all of the GPGPU methods that follow.

6.5 Comparing GPGPU Approaches

In this section, we explore the performance impacts of the modifications made to the GPGPU approaches in order to handle variable length documents. These modifications primarily made greater use of the GPGPU's global memory vice the shared memory-centric approach to perform document relatedness calculations. From our analysis in Chapter 4, we know that the best GPGPU loading method is the tuned stride approach and that the best data structure for looking up word similarities is perfect hashing. We now need to compare the following:

1. The shared memory method vs the global memory method
2. Pair-Wise document comparison approach vs the One-To-N document comparison approach

This leads to the evaluation of a total of four approaches. Each of the proposed GPGPU approaches were evaluated using the ACM Dalhousie Abstracts corpus in a 2,000:2,000 document comparison and in a 10,000:10,000 document comparison to best identify the impacts of the two memory methods on the selected approaches.

6.5.1 Evaluating the Shared Memory GPGPU methods

Table 6.1 illustrates the performance observed in the shared memory GPGPU approach.

GPGPU Approach	WPS	DPS	Comparison (N:N)
Shared Memory Pair-Wise	10,803,419	9,698	2,000:2,000
Shared Memory One-to-N	40,519,426	36,376	2,000:2,000
Shared Memory Pair-Wise	12,912,885	9,415	10,000:10,000
Shared Memory One-to-N	19,526,691	14,032	10,000:10,000

Table 6.1: Performance of the shared memory GPGPU approaches on 2,000:2,000 and 10,000:10,000 Document Relatedness Comparisons Using The ACM Dalhousie Abstract Collection

Based on the results observed in Table 6.1, Table 6.2 was created to easily illustrate the change in the performance of the shared memory approaches as the volume of the document relatedness calculations was increased.

GPGPU Approach	Δ WPS	Δ DPS
Shared Memory Pair-Wise	20%	-3%
Shared Memory One-to-N	-53%	-60%

Table 6.2: Contrasting The Shared Memory GPGPU Approach Performance Evaluating the ACM Dalhousie Abstract Collection for 2,000:2,000 against 10,000:10,000

Table 6.2 clarifies that the shared memory Pair-Wise approach scales effectively, demonstrated by the 3 percent decline in the reported DPS performance as the amount of work to compute the N:N document relatedness is multiplied by 25. This figure illustrates that the number of words compared between the $2{,}000^2$ documents and

$10{,}000^2$ documents increases by roughly the same factor as the total volume of documents compared. Additionally, Figure 6.1 allows us to observe that the corpus of $10{,}000^2$ document contains longer documents (approximately 10% longer). These two figures allow us to effectively interpret an increase in the rate of WPS.

In contrast to the shared memory Pair-Wise approach, the shared memory One-to-N approach has not demonstrated an ability to scale as effectively. The shared memory One-to-N approach shows a decline of 60% in terms of DPS, and as a consequence of lowered document throughput, the WPS rate declined accordingly.

In considering the rationale for the performance drop in the shared memory One-to-N approach, the GPGPU's ability to assign resources must be examined. As discussed in Section 2.3.4, the GPGPU assigns resources by allocating blocks, each of which is assigned a pool of threads. The N in the One-to-N approach defines both the number of blocks and documents to compare against a base document. Each of the N blocks is assigned a pool of threads equal to $|Document_{longest}|$ of the N documents being compared. As N increases, the amount of work required of the kernel also increases, but the number of resources remains fixed until N thread pools are finished.

This results in a performance bottleneck, where the GPGPU is unable to advance to the next kernel until the slowest pool of threads has completed its work. As the thread pools finish performing their document relatedness calculations and are removed from the GPGPU's processing queue, more and more of the GPGPU falls idle. In the One-to-N approach, this is caused by two issues:

1. Larger Documents require more iterations over the algorithm to calculate relatedness, slowing document throughput.

2. More work is provided upfront but no replacement tasking is available until the N documents are compared.

In contrast, the Pair-Wise approach saturates the GPGPU with at most 32 times the $|document_{smallest}|$ blocks of thread pools, each of which is required to perform less work than a corresponding kernel launched in the One-to-N approach. It also provides a more voluminous pool of tasks that can be worked on while waiting for slower (longer) documents to be processed.

Based on the Shared Memory limitations discussed in Section 4.2, we can conclude that the Shared Memory approaches cannot scale to a larger corpus, such as the Gutenberg Collection. On smaller corpora that would apply to this method, such as the ACM Dalhousie Collection, we would expect the performance of the One-to-N approach to continue to decline comparatively to the Pair Wise due to the limited number of threads it can effectively exploit on the GPGPU, as the scope of N in the N:N document relatedness increases.

6.5.2 Contrasting the GPGPU Global Memory Approach with the Shared Memory Approach

Table 6.3 illustrates the performance observed in the global memory GPGPU approach, when applied to the 2,000:2,000 and 10,000:10,000 ACM Dalhousie Abstract Collection.

GPGPU Approach	WPS	DPS	Comparison (N:N)
Global Memory Pair-Wise	8,154,287	7,320	2,000:2,000
Global Memory One-to-N	28,442,911	25,535	2,000:2,000
Global Memory Pair-Wise	9,975,764	7,273	10,000:10,000
Global Memory One-to-N	30,240,353	22,049	10,000:10,000

Table 6.3: Performance of the Global Memory GPGPU approaches on a 2,000:2,000 and 10,000:10,000 Document Relatedness Comparisons Using The ACM Dalhousie Abstract Collection

Based on the results observed in Table 6.3, Table 6.4 was created to easily illustrate the change in the performance of the global memory approaches as the volume of the document relatedness calculations increased.

GPGPU Approach	Δ WPS	Δ DPS
Global Memory Pair-Wise	22%	-1%
Global Memory One-to-N	6%	-14%

Table 6.4: Comparing the Rates of the ACM Dalhousie Abstract Document Relatedness Performance over 2,000:2,000 and 10,000:10,000 for the Global Memory GPGPU Approaches

Table 6.4 allows us to observe that the global memory Pair-Wise approach again demonstrates the most efficient relative scaling of DPS and WPS performance. The

global memory One-to-N approach, when contrasted with the shared memory One-to-N approach (see Table 6.2), shows a much more graceful degradation in performance in terms of DPS (14% decline), and even shows an increased throughput in WPS. This can in part be explained by the growth in the corpus' average document length, and scaled by the decline in the DPS.

The modifications made to the global memory approaches result in a clear improvement over the observed scaling performance of shared memory approaches. The superior scalability of global memory approaches are explained by the fragmentation of the work into smaller elements. In the shared memory approaches there were fewer kernel calls, and due to the modifications required for variable length documents, the global memory approaches alter the base algorithms to perform the same work, spread out over more kernel calls.

Prior to contrasting the performance of the global memory and shared memory approaches on the ACM Dalhousie Abstract Collection, one must first discuss the expected performance of the approach independent of memory storage. In general, if algorithms are equal, the One-to-N approach should always outperform the Pair-Wise approach. This is due to the GPGPU global memory access patterns for the algorithms, which are the most expensive operations performed on the GPGPU. The access patterns are as follows:

1. For the One-to-N approach **O**(N+M), where N is the length of the largest document, and M is the length of smallest of the two documents being compared.

2. For the Pair-Wise approach **O**(N*M), where N is the length of the largest document, and M is the length of smallest of the two documents being compared.

As each of the two approaches have the above **O**(memory access times) for each of the kernels they invoke, it follows that the global memory approaches of the algorithms should be slower for all approaches.

The global memory Pair-Wise approach performs slower than the shared memory Pair-Wise approach across the evaluation corpora, which is in line with expectations due to the fragmentation of the work. It also performs slower than the shared memory One-To-N approach on the sample sizes selected, though based on the discussion in Section 6.5.1, this condition most likely would not hold if the relatedness of a greater

number of documents was evaluated.

The global memory One-to-N approach, however, is only in line with the expected performance in smaller subsets. In the larger corpus, it out-performs the shared memory approach by a significant margin. While the global memory One-to-N approach performs with the highest WPS rate with the larger document collection, the performance can be explained by the following factors:

1. The shared memory approach had N bounded by the upper dimensions of the N:N document relatedness being calculated. The modifications required for variable length documents ensure that the N in the One-To-N approach is now bounded to $N \leq 64$ for the global memory approach. This dedicates a fixed volume of work for each iteration resulting in increased kernel launches over larger document sets.

2. Rather than the shared memory approach's single kernel, whose workload grows in relation to N, the global memory approach spread the single kernel's processing work over four kernels. This allowed for the kernels to be implemented so that there would be a lowered rate of conditional branching vice the single kernel of shared memory.

6.5.3 Evaluating the Global Memory GPGPU Approach

To determine the performance of the global memory on the GPGPU, each of the approaches, Pair-Wise and One-To-N, were evaluated with global memory. This was performed with a series of subsets of varying sizes, from each corpus.

Figure 6.7 illustrates the observed performance of the global memory Pair-Wise and One-to-N approaches when calculating the relatedness for the ACM Dalhousie Abstract Collection. We observe in this figure that the words compared per second is a function of the number of documents compared. We observe that as the number of documents compared increases, the rate of WPS increases as well. This figure shows that the One-to-N approach is clearly superior to the Pair-Wise approach on smaller data sets, and that the One-To-N approach has a rate of 30 million WPS, while the Pair-Wise approach has a rate of 10 million WPS. The rationale for the performance of the One-To-N approach can be explained by:

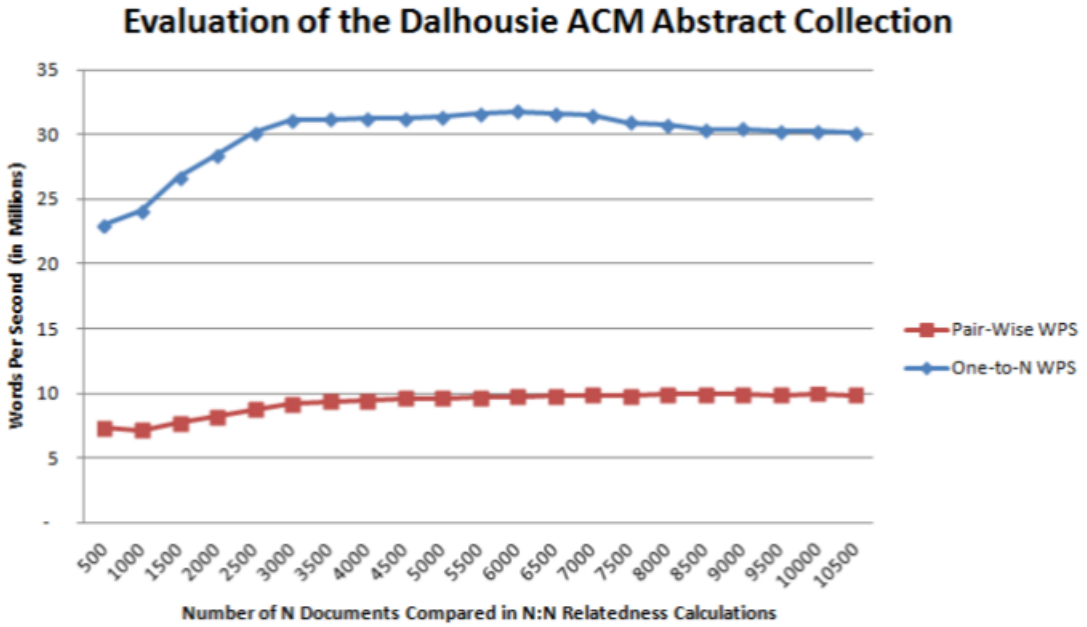

Figure 6.7: The Observed Performance of the Global Memory Approaches When Processing Segments of the ACM Dalhousie Abstract Collection

1. More efficient document transfers from the CPU to the GPGPU.

   The Pair-Wise approach requests a maximum of 32 individual document transfers at a time, while the One-to-N requests up to 64 at once. This allows for less overhead for the transfer of the data across the Peripheral Component Interconnect Express (PCIe) data bus.

2. Smaller documents.

   The smaller documents allow for the workload of the One-to-N approach to be more efficiently assigned than with the Pair-Wise approach. Expected performance of the One-to-N approach is **O**(N+M), due to performing the iteration over the smaller documents while holding the larger documents in thread memory. On smaller documents there are less iterations and therefore a faster throughput.

Figure 6.8 illustrates the observed performance of the global memory Pair-Wise and One-to-N approaches when calculating the relatedness for the Gutenberg Collection.

We observe in this figure that words compared per second (WPS)is a function

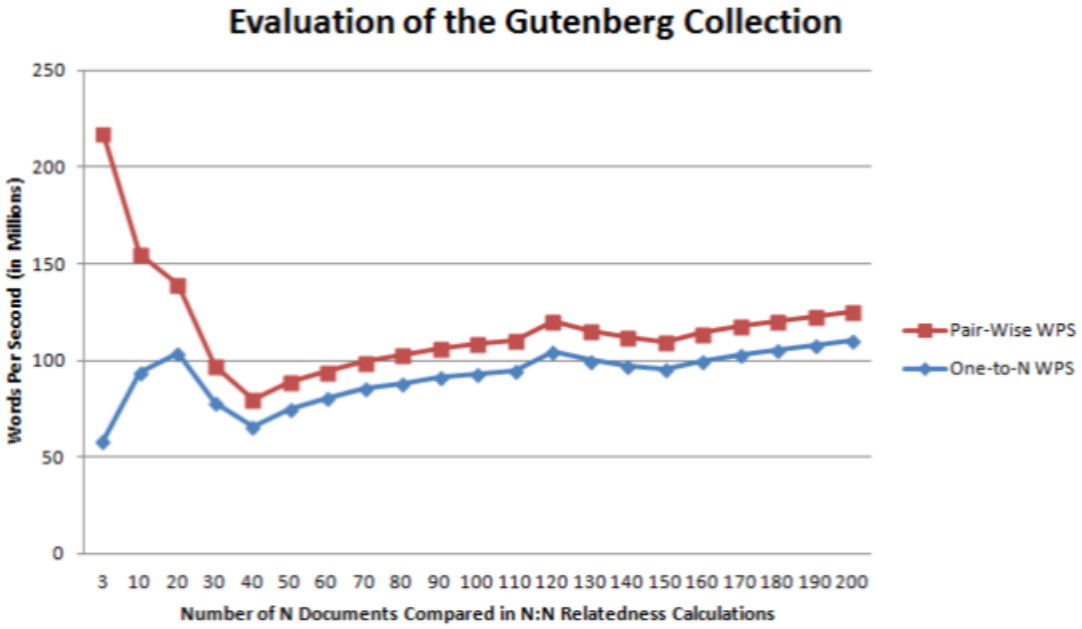

Figure 6.8: The Observed Performance of the Global Memory Approaches When Processing Segments of the Gutenberg Collection

of the number of documents compared. To this effect, as the number of documents compared increases, the rate of WPS increases as well. This figure shows that the Pair-Wise approach outperforms the One-to-N approach, and that the One-To-N approach has a rate of 110 million WPS, while the Pair-Wise approach has a rate of 130 million WPS. The rationale for this performance difference can be explained by:

1. Document transfers from CPU to the GPGPU are non-blocking.

   The Pair-Wise approach requests at most 32 individual document transfers at a time, and the relatedness calculation for a document can begin as soon as the first transfer is completed.

2. Larger documents.

   The larger documents allow for the Pair-Wise approach to scale properly. Rather than the One-to-N approach of iterating over the smaller document of the pair, the Pair-Wise approach spawns more threads to represent the document. While this approach would create more threads than relative work required on smaller documents, with larger documents the threads are provided with enough work.

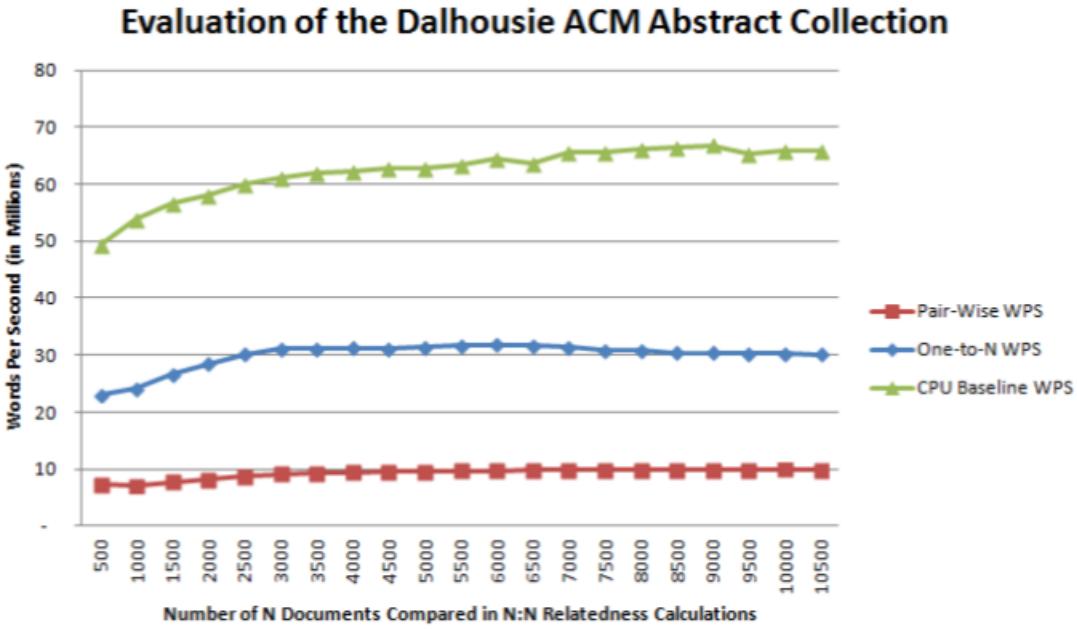

Figure 6.9: Performance of the ACM Dalhousie Abstract Collection Across All Approaches

Ultimately, the selection of the most effective approach is decided based on the content and complexity of the corpus. The One-to-N approach provides the more generally applicable approach to document relatedness calculations. If the documents of the corpus are sufficiently large, for example six thousand unique words, then the Pair-Wise approach would be the preferred approach.

6.6 Comparing Global Memory GPGPU Approaches to the Benchmark

In this section, we compare the two GPGPU approaches, using global memory, to the CPU benchmark when performing a document relatedness calculation.

Using the benchmark established in Section 6.4, the global memory GPGPU approaches can be now be evaluated. This evaluation compares the baseline against the GPGPU approaches on the ACM Dalhousie Abstract Collection and the Gutenberg Collection.

The approaches were evaluated using a series subsets of composed from the first 10,500 documents of the ACM Dalhousie Abstract Collection. The performance of the approaches with this subset is shown in Figure 6.9.

Figure 6.9, illustrates the WPS rates of the optimal approaches with the ACM

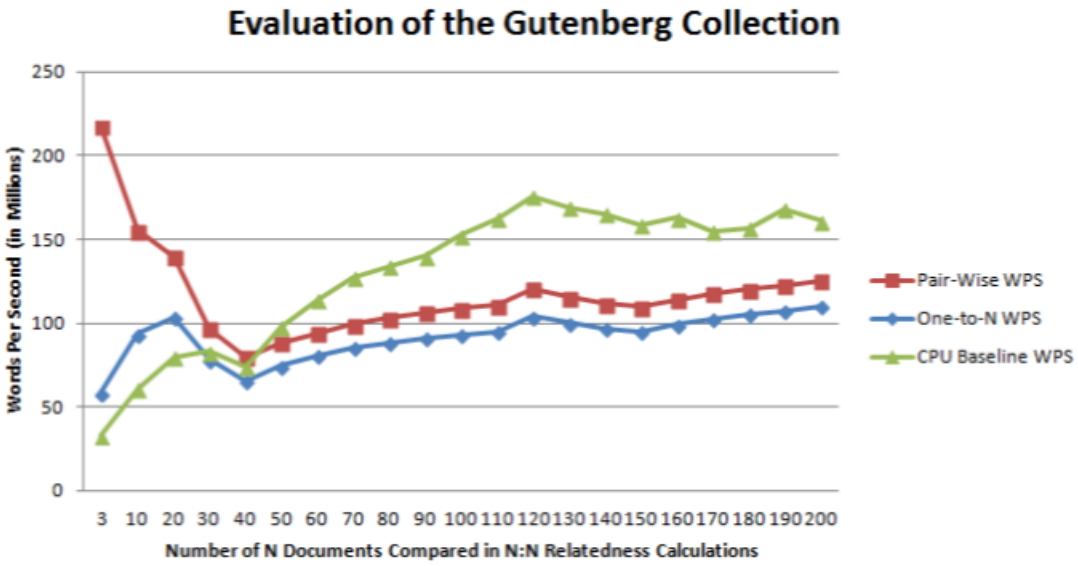

Figure 6.10: Performance of the Gutenberg Collection Across All Approaches

Dalhousie Abstract Collection: the global memory Pair-Wise approach, the global memory One-To-N approach, and the CPU benchmark. We observe that the CPU benchmark performs with a WPS rate of around 65 million, easily out-performing the closest GPGPU approach. The One-to-N approach performed the best out of the GPGPU approaches, with a WPS rate of around 30 million, while the Pair-Wise had an observed WPS rate of 10 million. We can infer, based on the performance of the CPU benchmark, that the streaming of the ACM Dalhousie Abstract Collection's smaller documents is responsible for some of the reduced capacity of the GPGPU. This explanation is supported in part by the performance of the One-to-N approach, with its larger document transfer performing significantly better relative to the CPU benchmark.

Figure 6.10, shows the WPS rate of the three methods being evaluated with the Gutenberg Collection: the global memory Pair-Wise approach, the global memory One-to-N approach, and the CPU benchmark. We observe that the One-to-N approach performs with a relatively consistent WPS rate of a 92 million, which initially out-performed the CPU benchmark, until 30^2 documents were compared, at which point the CPU benchmark demonstrated superior performance. The Pair-Wise approach performed with an average WPS rate of 120 million, outperforming the One-To-N approach under all conditions. The Pair-Wise method outperformed the CPU

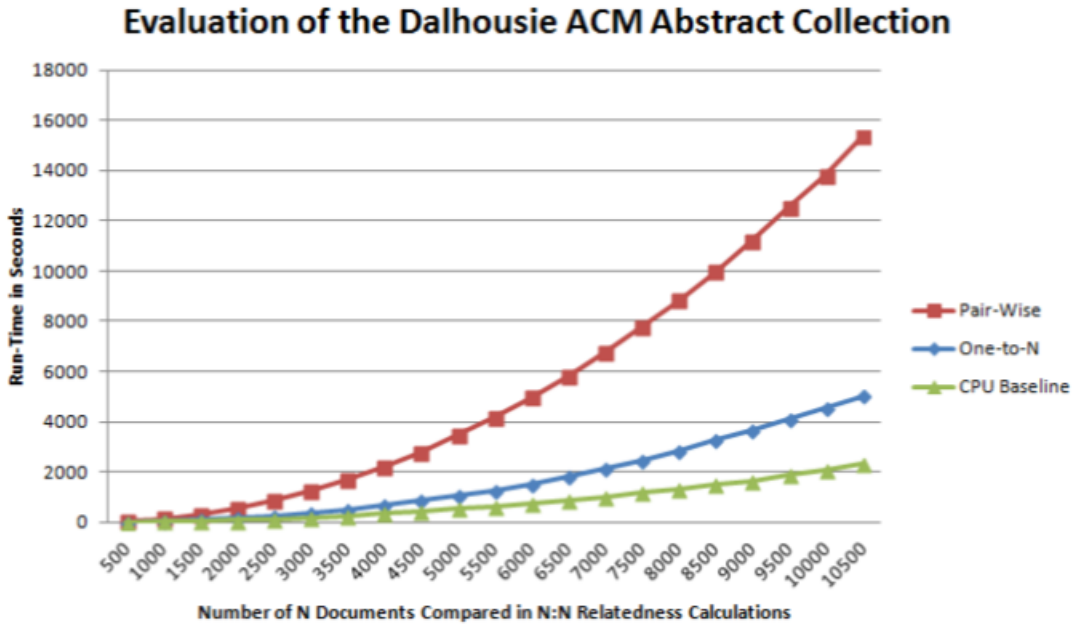

Figure 6.11: Run-Time Performance of the ACM Dalhousie Abstract Collection Across All Approaches

benchmark until 50^2 documents were compared at which point the CPU benchmark outperformed the Pair-Wise approach.

Figure 6.11 illustrates the run-times of the Pair-Wise, One-to-N and CPU benchmark approaches when evaluated on the same subsets of the ACM Dalhousie Abstract Collection. In Figure 6.9, we observe that the Pair-Wise approach takes the most amount of time to calculate the results, while the One-to-N approach performs the same volume of work in a similar amount of time as the benchmark until more than $3{,}000^2$ documents are compared.

Figure 6.12, illustrates the run-times of the Pair-Wise, One-to-N and CPU benchmark approaches when evaluated on the same subsets of the Gutenberg Collection. In Figure 6.10, we observe that the One-to-N approach takes the most time to calculate the results, while the Pair-Wise approach performs the same volume of work as the benchmark until more than 60^2 documents are compared.

Based on the results observed in this evaluation, the GPGPU approach can only out-perform the benchmark for limited intervals and only under specific conditions. To narrow the performance gap between the GPGPU and CPU, arguments can be made that improving the hardware of the GPGPU used, for instance, updating the

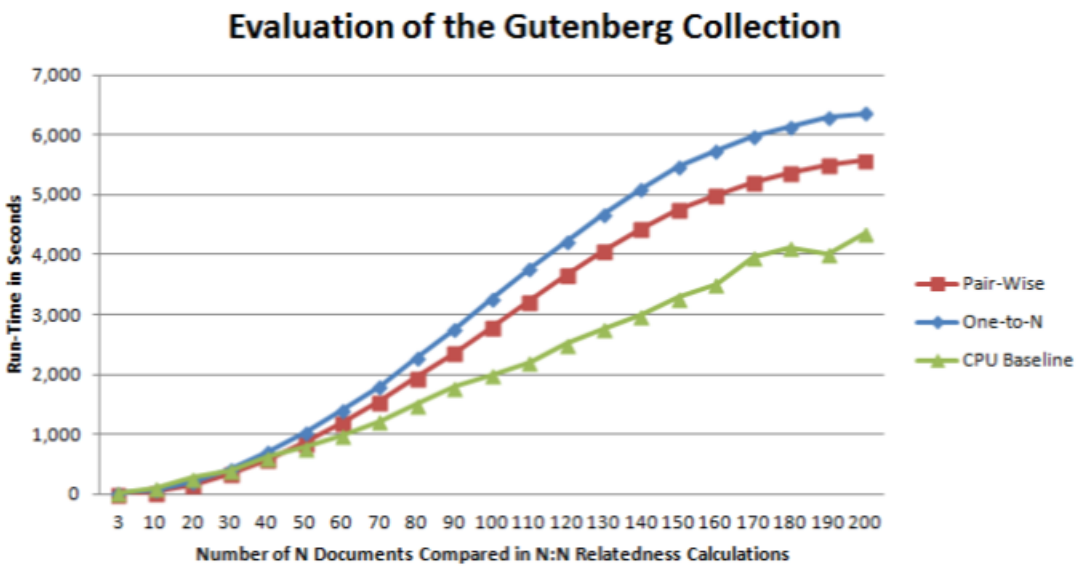

Figure 6.12: Run-Time Performance of the Gutenberg Collection Across All Approaches

GeForce 660 GTX to a more recent video card. However, a similar argument could be made for improving the server that the benchmark is run on. This is to say that the specific hardware utilized can negatively or positively affect the word relatedness calculations on either a CPU or a GPGPU.

To contextualize the GPGPU performance in terms of the CPU performance, one should consider the financial or retail cost of the hardware. In this series of experiments, the GPGPU is a sub $200 card compared against CGM6, an $8,000 server. In this case, the superior performance of CGM6 comes at 40 times the price of the GPGPU.

6.7 Summary of Results

To summarize the results of this chapter's experiments, the GPGPU approaches have been shown to effectively calculate document relatedness given the conditions that were outlined in Section 3.1.

Based on the performance observed during the document similarity computations on the documents, the optimal approach between the Pair-Wise and the One-to-N approaches is determined by the average corpus length. On relatively small corpora, such as the ACM Dalhousie Abstract Collection, the One-to-N approach demonstrates

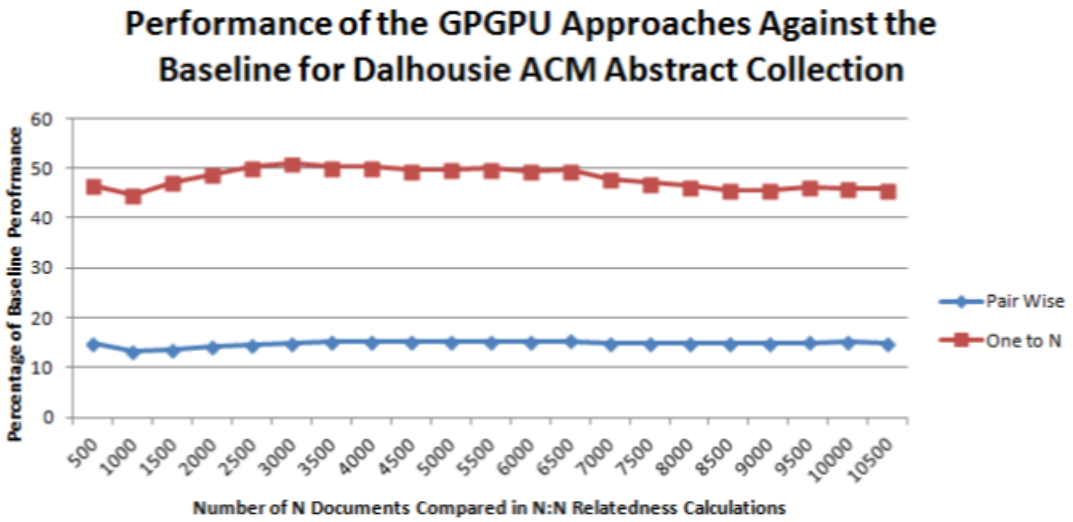

Figure 6.13: Performance of the Global Memory GPGPU Approaches on the ACM Dalhousie Abstract Collection Expressed as Percentage of the Baseline Performance

superior performance. Given the relatively small number of data sets of this size, the Pair-Wise approach would be more universally applicable, and therefore, the generally preferred approach to use.

Figure 6.13 illustrates the performance of the global memory One-to-N, and global memory Pair-Wise approaches on the subset of the ACM Dalhousie Abstract Corpus used earlier in this chapter. The performance of the two approaches are expressed as a percentage of the CPU benchmark's WPS performance. This allows us to observe that the One-to-N approach provides the performance of about half of the CPU benchmark (while the Pair-Wise approach sits at 15%).

Figure 6.14 illustrates the performance of the global memory One-To-N, and global memory Pair-Wise approaches on the subset of the Gutenberg Collection used earlier in this chapter. The performance of the two approaches are expressed as a percentage of the CPU benchmark's WPS performance. This allows us to observe that the One-to-N approach provides the performance of around 60% of the CPU benchmark, with the Pair-Wise approach performing slightly higher at around 80%.

Ultimately, this thesis has proven the ability of the novel GPGPU approaches proposed to compute document relatedness, and has illustrated that the performance of a single low-cost GPGPU can vary between 40 to 80% of a high-cost mutli-core server, in this thesis CGM6, as illustrated in Figures 6.13 and 6.14.

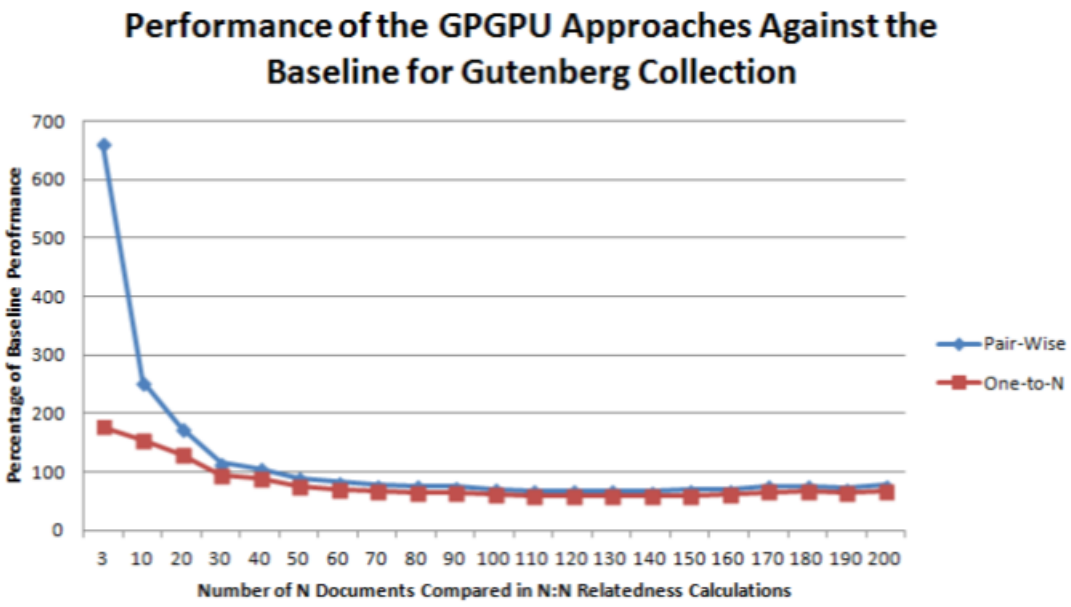

Figure 6.14: Performance of the Global Memory GPGPU Approaches on the Gutenberg Collection Expressed as Percentage of the Baseline Performance

Chapter 7

Conclusion

This thesis presented and evaluated a number of approaches for computing GTM relatedness on a GPGPU platform. It has illustrated that document relatedness calculations can be effectively performed on GPGPUs in a highly cost effective manner. In addition to proving the success of the novel GPGPU approaches as an alternative to previously established CPU approaches, this thesis has illustrated that the observed performance of a single low-cost GPGPU can compete with the performance of a high-cost mutli-core server.

7.1 Recommendations for Future Work

While the results presented in thesis have demonstrated an effective GPGPU GTM approach, the approach could be further developed in a number of ways. For example, the following areas of future development could be explored:

1. A Hybrid CPU and GPGPU algorithm to conduct GTM document relatedness.

   This approach would require investigations into scheduling and assigning work between the CPU and GPGPU approaches. This would also include devising an improved CPU implementation of GTM.

2. An adaptive GPGPU GTM method that selects the algorithmic approach of Pair-Wise or One-to-N, based on the corpus being evaluated.

   By performing additional research into the performance of the Pair-Wise and One-to-N approaches on corpora of varying document lengths, a heuristic for which approach to use given set a documents can be developed. Using this heuristic, the GPGPU GTM method could then determine and apply the best suited approach for calculating relatedness to achieve the highest rate of WPS possible.

3. A multi-GPGPU approach to performing GTM document relatedness.

   The multi-GPGPU approach would require experiments to determine the ideal methodology to share workload, and resources between multiple GPGPUs.

4. Further analysis into other applicable N:N document comparison methods, such as scheduling blocks of documents.

   This thesis explored 1:1, and 1:N relatedness approaches to produce an N:N document relatedness calculation, and while effective, there are other relatedness approaches explored for multi-core implementations [15, 19]. This future work would involve the investigation and application of these approaches to a GPGPU.

5. Applying the GPGPU approach to other corpus-based similarity methods.

   The GPGPU approach in this thesis can potentially be applied to other methods to compute document similarity, such as work presented in [14].

Bibliography

[1] Jon Martindale, Nvidia dominating in add-in graphics card market, DigitalTrends, Aug. 2015. [Online]. Available: http://www.digitaltrends.com/computing/nvidia-add-in-graphics-market/ [Accessed: 27 Aug. 2015].

[2] Bob Jenkins, Minimal Perfect Hashing, BURTLEBURTLE, 1999. [Online]. Available: http://burtleburtle.net/bob/hash/perfect.html [Accessed: 20 June 2014].

[3] Thorsten Brants and Alex Franz. Web 1T 5-gram corpus version 1.1. Technical report, Google Research, 2006.

[4] Fellbaum. WordNet: An electronic lexical database. *MIT Press*, 1998.

[5] Wael H. Gomaa and Aly A. Fahmy. A Survey of Text Similarity Approaches. *International Journal of Computer Applications*, 2013.

[6] Anna Huang. Similarity measures for text document clustering. 2008.

[7] Aminul Islam and Diana Zaiu Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *TKDD*, 2(2), 2008.

[8] Aminul Islam, Evangelos E. Milios, and Vlado Keselj. Comparing word relatedness measures based on google n-grams. In *COLING 2012, 24th International Conference on Computational Linguistics, Proceedings of the Conference: Posters, 8-15 December 2012, Mumbai, India*, pages 495–506, 2012.

[9] Aminul Islam, Evangelos E. Milios, and Vlado Keselj. Text similarity using google tri-grams. In *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, pages 312–317, 2012.

[10] Jason Sanders, Edward Kandrot. *CUDA By Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[11] Shibamouli Lahiri. Complexity of Word Collocation Networks: A Preliminary Structural Analysis. *ArXiv e-prints*, 2013.

[12] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 451–460, 2010.

[13] Hongzhe Liu and Pengfei Wang. Assessing sentence similarity using wordnet based word similarity. *JSW*, 8(6):1451–1458, 2013.

[14] Md Rashadul, Hasan Rakib, Aminul Islam, and Evangelos Milios. Text relatedness using word and phrase relatedness. *Proceedings of the 9th International Workshop on Semantic Evaluation*, 2015.

[15] Jie Mei, Xinxin Kou, Zhimin Yao, Andrew Rau-Chaplin, Aminul Islam, Abidalrahman Moh'd, and Evangelos E. Milios. Efficient computation of co-occurrence based word relatedness. In *Proceedings of the 2015 ACM Symposium on Document Engineering, DocEng 2015, Lausanne, Switzerland, September 8-11, 2015*, pages 43–46, 2015.

[16] Donald Metzler, Susan T. Dumais, and Christopher Meek. Similarity measures for short segments of text. pages 16–27, 2007.

[17] Gabriel Recchia and Max M. Louwerse. A comparison of string similarity measures for toponym matching. In *ACM SIGSPATIAL International Workshop on Computational Models of Place, COMP 2013, November 5, 2013, Orlando, Florida, USA*, pages 54–61, 2013.

[18] Robert B Allen, Pascal Obry, and Michael Littman. An interface for navigating clustered document sets returned by queries. pages 1075–1079, 1993.

[19] Xinxin (Vivian) Kou. Efficient Implementations of Google Trigram Method for Computing Document Relatedness. Bachelor thesis, Dalhousie University, 2015.