Accelerating Web Search using GPUs

by

Rimon Tadros

PGD., Alexandria University, 2007
B.A.Sc., Alexandria University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2015

Abstract

The amount of content on the Internet is growing rapidly as well as the number of the online Internet users. As a consequence, web search engines need to increase their computing capabilities and data continually while maintaining low search latency and without a significant rise in the cost per query.

To serve this larger numbers of online users, web search engines utilize a large distributed system in the data centers. They partition their data across several hundred of thousands of independent commodity servers called Index Serving Nodes (ISNs). These ISNs work together to serve search queries as a single coherent system in a distributed manner. The choice of a high number of commodity servers vs. a smaller number of supercomputers is due to the need for scalability, high availability/reliability, performance, and cost efficiency.

For the web search engines to serve a larger data, the web search engines can be scaled either vertically or horizontally [21]. Vertical scaling enables ranking more documents per query within a single node by employing machines with higher single thread and throughput performance with bigger and faster memory. Horizontal scaling supports a larger index by adding more index serving nodes at the cost of increased synchronization, aggregation overhead, and power consumption.

This thesis evaluates the potential for achieving better vertical scaling by using Graphics processing unit (GPUs) to reduce the documents ranking latency per query at a reasonable initial cost increase. It achieves this by exploiting the parallelism in ranking the numerous potential documents that match a query to offload to the GPU. We evaluate this approach using hundreds of rankers from a commercial web search engine on real production data. Our results show an 8.8× harmonic mean reduction in the latency and 2× power efficiency when ranking 10000 web documents per query for a variety of rankers using C++AMP and a commodity GPU.

Preface

This dissertation is original, unpublished, independent work by the author, Rimon Tadros

Table of Contents

List of Tables

List of Figures

Glossary

FE Feature Extraction

FSM Finite State Machine

FTP File Transfer Protocol

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Units

HLSL High-Level Shader Language

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IDF Inverse Document Frequency

IR Information Retrieval

ISN Index Server Node

ISR Index Reader

L1Ranking Level 1 ranking also called selection

L2Ranking Level 2 ranking also called ranking

MIE Model Input Evaluation (FE and MIT)

MIT Model Input Transformation

ME Model Evaluation

QPS Query per second

SE Search Engine

SERPs Search Engine Result Pages

SLA Service Level Agreement

UI User Interface

URL Uniform Resource Locator

WSQ Web Search Query

WWW World Wide Web

Acknowledgements

I would like to thank my supervisor, Professor Tor Aamodt, for all the help in this great learning journey. Also, my managers and peers at Microsoft who supported me to finish this work. I would also like to thank the other members of the computer architecture group for their support. Finally, I'd like to thank my family for everything that they have helped me out with throughout my life.

Chapter 1

Introduction

Web Search engines are the systems that help users to find relevant web pages on the Internet quickly. The users enter a few keywords, and the search engines return a page that contains the result as a list of links to relevant pages on the Internet. The result relevant pages are not similar to each other. In another word the search engines return relevant pages from multiple prospectives that called "diversification" [1].

Web search engines are crucial to the Internet users. Commonly, the Internet users reach web pages on the Internet by one of the following ways; using web search engines, following links from page to page, or typing the URL in the browser. [34] shows that the web search engines influence 13.6% of the users web traffic and helps users reach 20% more sites. Other surveys show that search affect 50-80 percent of the way users find a new web page.

Web Search engines are large and competitive business and to attract more users, they must continuously expand their compute capability and data to match the increasing size of the Internet and the growing number of users. The size of the internet is growing exponentially [14, 16, 24, 43]. There are many techniques to measure the size of the Internet including the Monte Carlo and the importance sampling [43]. The number of the Internet users are increasing linearly, and reached 3 billions of users [40]. The rise of the amount of content on the Internet and the growth in the number of users are challenging for the web search engine. This challenge has led researchers and practitioners to look at software and hardware approaches to improve the system scalability.

The core data of the web search engines is its index. The web search index can be simplified as a record of where do words appear on the Internet web pages. In reality, there are types of indexes, and they contain much more information about the web pages that they index. The index size is crucial to a better user experience. The index size is measured by the number of web pages indexed in that index.

The Internet is large, and any commercial web search engine index will always contain a subset of the information on it. The size of the major web search engines index is not public information. However, some effort like

in [6] shows that the index size of Google is around 50 billions of pages while the index size of Bing is around 15 billion of pages. The larger the web search index, the more information it has that can be used to satisfy the users queries that lead to a better user experience. The index needs to be periodically updated to match the changes of the internet. Index high freshness and Index volume increase are significant challenges. Expanding the index is challenging in both stages; index building (also called indexing) and index serving (serving the users queries from the index).

Creating and growing the index requires crawling, indexing, compression, and storage. Crawling is following the URL links from one page to other pages, also called the web spider, then downloading these pages. Indexing is recording what words appear in the crawled pages. Commercial web search engines use distributed index building systems that run on multiple servers simultaneously. There are many challenges in crawling including spam detection, content quality measuring, duplicates removal, network optimization, and scalability. Scalability in these steps is challenging. Still, they are offline processes because they happen in the background without direct interaction with the end user. Therefore, the index building has more relaxed restrictions. Because of that, we decided to look into the scalability of the most challenging problem that is the index serving scalability.

Serving the live users queries is an online process. Query processing that services a bigger index is challenging in many ways such as in index reading, decompression, caching, documents selection, and document ranking. The index reading and caching are I/O bound processes. Document Selection is the process of finding web pages that contain the user query words. While document ranking is ranking these selected pages to present the high ranked documents to the user. The higher the document rank, the higher the relevance to the user query. These steps, index reading, decompression, caching, documents selection, and ranking, happen in a very restricted time budget or SLA (Service level agreement).

Users are expecting to get their search result in less than a second. Studies show that users who experience a delay in the response of any website get frustrated and tend to leave and ignore that site [5, 9]. Therefore with this fixed latency budget per query, any improvement in one process of the query processing enables more time to be available for other processes and reduces the overall query latency. The decrease of the query latency leads to happier users. Also, another benefit of speeding up the query processing is that it enables searching through a bigger index per node. Searching in larger index fast increases the search quality that also leads to satisfied users.

Improving the scalability by employing more powerful servers or by in-

creasing the number of servers always comes at the cost of more power consumption. Power consumption in the data center incurs the majority of the cost. The data center energy consumption is growing. Some studies [17] suggested that growth in the data center electricity use from 2005 to 2010 is 50%. Recent research [15] explored using less power consuming cores to reduce the cost of query processing. While energy was reduced, this came at the expense of decreased quality of search results since fewer documents could be ranked within this fixed latency budget. There is a need to use efficient hardware that can deliver better query/$. Researchers looked at using custom fabric that delivers a more efficient solution than the general purpose CPU [33], but it comes at the cost of development effort and code maintainability. In this thesis, we consider accelerating document selection and ranking on GPUs, using a programming language similar to C++ called C++AMP. C++AMP is shown to increase the development productivity by providing elegant ways to handle the users kernel code and synchronizing the data between the CPU and the GPU memory. The users kernel code uses lambda expression inside the parallel_for_each and restrict(amp) keyword that allow the developer to write code efficiently. Moreover, the data copy between the CPU and GPU memory is handled using the array and array_view constructs that make the data transfer transparent to the developer.

Also, we are using well-established development and debugging tools such as Microsoft Visual Studio IDE and development stack. Using the GPU enables ranking higher number of documents per query within this fixed latency budget at a better cost.

Graphics Processor Units (GPUs) such NVIDIA's Kepler [29] have been shown to increase throughput by up to an order-of-magnitude versus well optimized multicore CPU code [19]. This increase in the throughput is possible because they dedicate more area to computation rather than complex instruction scheduling hardware and cache.

1.1 Contributions

Below is the summary of our contributions:

- Evaluation of the use of GPUs in the data center workload. We examined the use of GPUs for web search engines query processing. Figure 2.4 shows the document ranking steps and the modules that we accelerated. We improved the fast first-pass rankers of the selection phase as well as the second-stage expensive rankers of the ranking

phase. These improvements lead to improving the overall performance of document ranking.

- One of the first evaluations of large-scale software on a GPU using C++AMP.

1.2 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides a summary of the web search architecture we aim to increase the index size of by using GPUs. Also, it presents a summary on the GPUs and their programming model.
- Chapter 3 describes the changes we made to our web search engine to offload part of the query processing onto a GPU using C++AMP.
- Chapter 4 describes our evaluation methodology.
- Chapter 5 provides our experimental results and analysis.
- Chapter 6 summarizes related work.
- Chapter 7 concludes.

Chapter 2

Background

In this Chapter, we will give a background on the web search engines and GPUs.

2.1 Review of Modern Web Search Engines

Web search engines are systems that index the Internet (called documents) to make its content searchable. Examples are Google and Bing. They allow users to search these indexes to find web pages on the Internet using a simple HTML UI. This simple web UI typically contains a search box that users type the keywords in and a result page that contains around 10 to 20 links to pages on the Internet. These links are called the search engine "blue links".

2.1.1 Web Search Engine Components

Figure 2.1 show the main subsystems of the web search engine and their relations together. These subsystems are index builder and index server. The Index Builder contains the crawler [13]. A crawler is a software that visit pages on the internet and download their content then follow the links to these pages to find more pages. Crawlers face many challenges including memory size, network optimization, and spam. There are an immense number of spam web pages on the Internet. Spam can generate millions of dollars if spammers get the users clicks. The spammers artificially generate the spam web pages, and they make them crawler friendly. The crawler has to detect the spam pages and add metadata to the index to differentiate these pages [27]. After downloading the web page, the Index Builder add, update, or delete the content of that page in the Index. This process of modifying the index is called index merge, and it happens with minimum interruption to the index serving process.

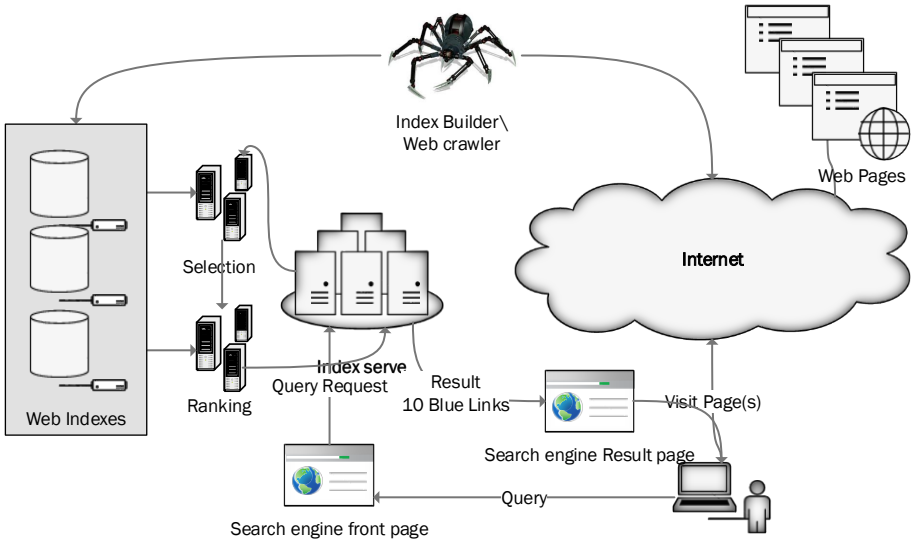

Figure 2.1: Overview of web search engine.

2.1.2 Web Search Engine Indexes

There are two main types of indexes; inverted indexes and forward indexes [20]. The Inverted Index is a map from words to web pages which means giving a word the inverted index outputs a list of web pages that contain that word. In an inverted index web pages on the internet is assigned a unique identifier called document identifier or DocId. An example to clarify how the inverted index works: The inverted index for two words "Lazy Dog" is as following:

- Lazy: 10, 12, 18, ...
- Dog: 2, 8, 12, 15, 18, ...

That means the word "Lazy" appeared in documents: 10, 12, and 18. Also, the word "Dog" appeared in documents: 2, 8, 12, 15 and 18. The document selection phase of "index serve" uses the inverted index to find the pages that contain a user's query words quickly. Finding the web pages that contain multiple keywords using the inverted index is called list intersection. The inverted index is compressed and stored on disk, and part of it gets cached in memory [44].

The forward index contains a map from pages to words. Each word has an identifier or WordId. An example of the forward index for DocId 10 and 15.

- DocId 10: 2, 5, 7, ...
- DocId 20: 7, 10, 7, ...

That means document id 10 has words id: 2, 5, 7, and so on. Also, document 20 has word 7, 10, 7, and so on. The forward index is used for feature extraction in the document ranking phase of index serving as we will see in the next sections.

These indexes are represented by bitmaps. They are compressed and stored on disk. At run time parts of these indexes are loaded and decompressed from disk and stored in memory.

2.1.3 Query Processing and Index Serving

Users search queries are handled by the index serving part of the web search engine. Figure 2.2 shows the targeted index serving workflow end-to-end system [4, 8, 15, 45]. Users enter the search query in the web browser that sends an HTTP request to the search engine. The search query gets routed to a certain data center by the load balancer. The search query comes to the "front end". It gets checked against a recent results cache. If the query is in the cache (cache hit), the query answer (search result) also exists in the cache. The index server returns the results to the user immediately. This cache reduces the search query cost because each hit in the cache brings the result to the user without performing an expensive index search. There are many optimizations that are built in the results cache to determine when to store a new query and when to evict existing queries. When the cache miss, the search query answer is not in the cache, the query request is passed on to the back-end. The back-end has a result aggregator. The result aggregator initiates a search request to a set of index serving machines (ISNs). This set of machines together contains a copy of one full index. Each machine works independently to search its portion of the index. The request is sent to the document selection phase. Document selection phase uses the inverted index, selects the candidate documents, and forward the ranking request to the Ranking stage.

Ideally, all the index serving machines reply back to the aggregator with the result (set of documents) and a corresponding score per document. However, in some cases some of the ISNs fail, and the query is not completed.

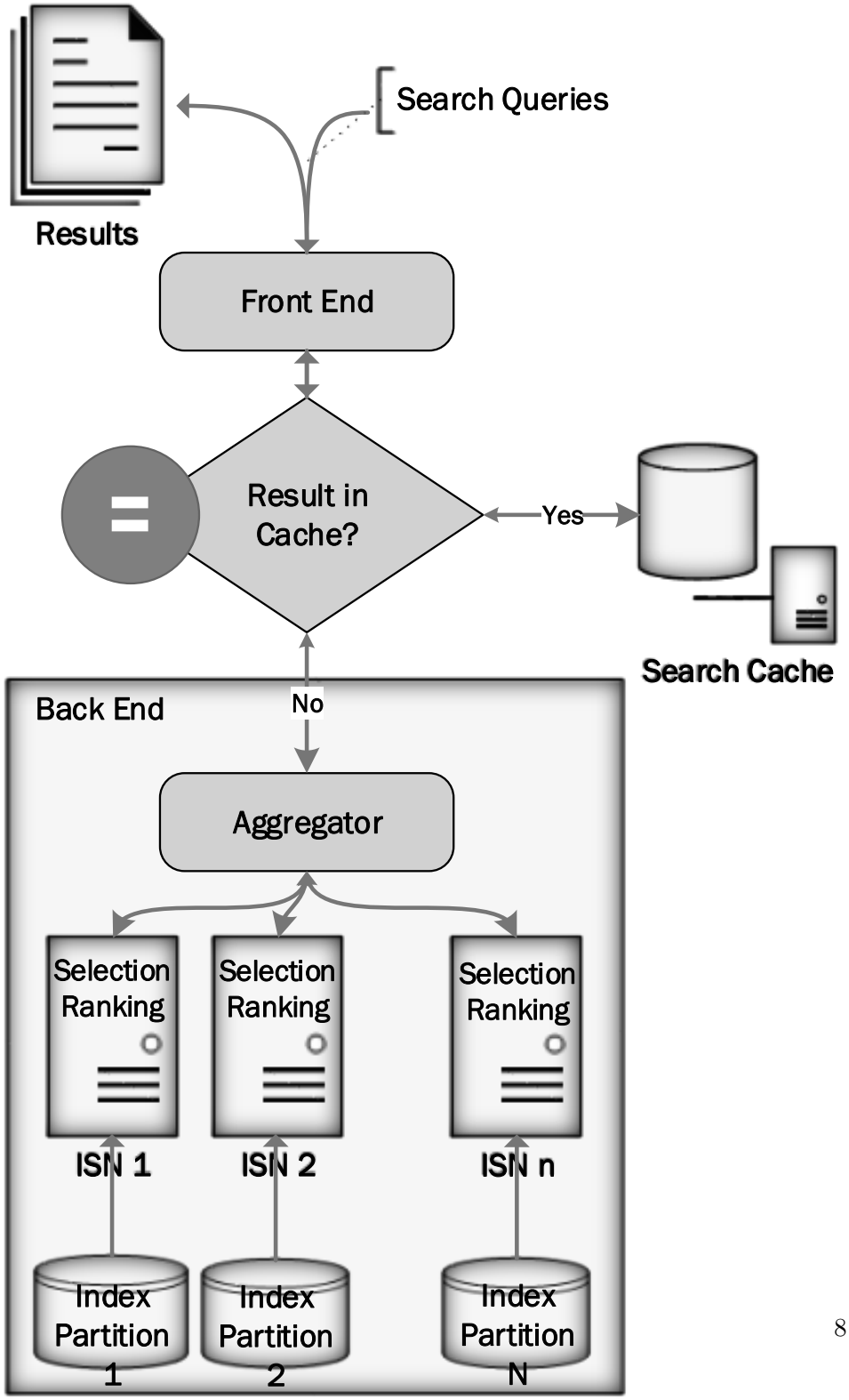

Figure 2.2: Overview of Index Serve.

In this case, the partial result is sent back to the user but it does not get cached. The higher the score, the more relevant the document is. ISNs must reply back to the aggregator within a certain duration called SLA "service level agreement". This rigid time restriction makes some ISNs terminate the search before ranking all pages in its partial index, known as early termination [46]. Early termination may affect the search quality. The aggregator merges and sorts the results coming from multiple ISNs, adds captions (text that appears under each result's web link), and forwards the result to the front-end. The front-end caches the result if the search was successful, and sends the result back to the user.

Scaling in these distributed architectures happens in two ways: vertical, and horizontal. Vertical scaling happens by obtaining more powerful/faster machines with more CPUs/cores, more memory, and a faster disk or SSD. The problem with this method of scaling is that multicore processor performance growth is not keeping up with the exponential growth of the Internet. Horizontal scaling happens by adding more index serving machines to this distributed system. Ignoring the aggregation overhead, the problem with this is the increase of each query cost, with the growing number of the physical hardware the data centers. The increase is twofold, first, because of the initial hardware cost and, more importantly, the operation cost that is added by the additional ISNs. For example, to scale from serving an index of size $1\times$ to $2\times$ with the same hardware, we need to double the number of index server machines. The query cost will double even for queries that have the best result (high relevant documents) in the smaller index and do not need that larger index. The query cost increase happens because each query that misses in the cache will use twice the number of machines after scaling. Typically search engines mitigate this by having different sized indexes. This optimization is out of the scope of this thesis.

2.1.4 Document Ranking Overview

Query processing involves two main steps [45]. Figure 2.3 describes these steps. The first step is the document selection step that utilizes a fast first-pass rankers. This step matches the indexed documents with the search query terms. Which also means it finds the set of documents that contain the query terms. This step works on a large number of documents; its input is the inverted index partition on the selection ISN and the query terms. The output of this step is a set of documents that contain all or some of the search query terms. The index generator compresses the inverted index [49] and stores it on disk/SSD. At runtime, the index manager loads

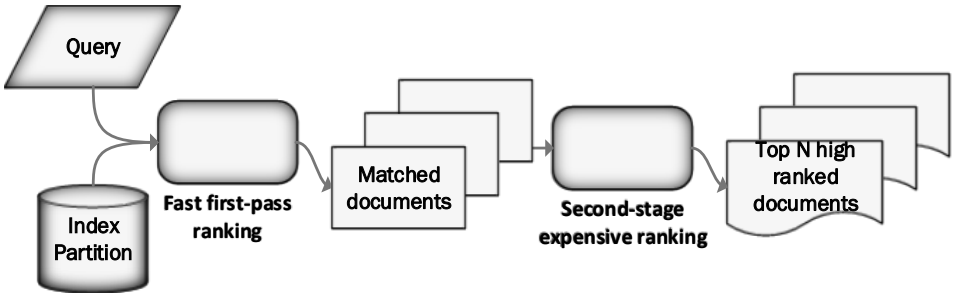

Figure 2.3: Query processing main steps.

parts of the index from disk/SSD, decompresses them, and caches them in memory. Interestingly the index cache hit rate can be up to 90% even if the memory holds only 20% of the inverted index at any given time [8]. The inverted index format maps general words to documents that contain these words. The inverted index makes it easy and fast to find the set of documents that contain the query words. In this fast first-pass ranker phase, the index server decompresses the index, finds a list of documents for each query word, does list intersection, and runs a simple scoring ranker on the resulting documents.

For example, a search for "Lazy Dog" in the example above, will return documents 12, 18 after the list intersection. After it will run a simple ranker on each document to give it a score. This simple scoring ranker needs to be fast in the current architecture because it runs on a large number of documents per index partition. The simple ranker in this phase requires some document features to be extracted (retrieved and/or calculated). Document features have two types: static and dynamic.

Static document features can be simply retrieved from the index (like the static rank or page rank [32], which reflects the quality of the document). Whereas, dynamic document features need to read more data from the document and run some calculations, as a number of word occurrences, BM25 [35], and other features.

The output of this fast first-pass ranking is a set of documents selected from the local ISN index partition that could potentially be a good match to the query. There are many techniques for inverted index compression/decompression and list intersection [49]. In our system, we run this process on the CPU. This output set of documents is passed to the next step, the expensive second-stage ranking.

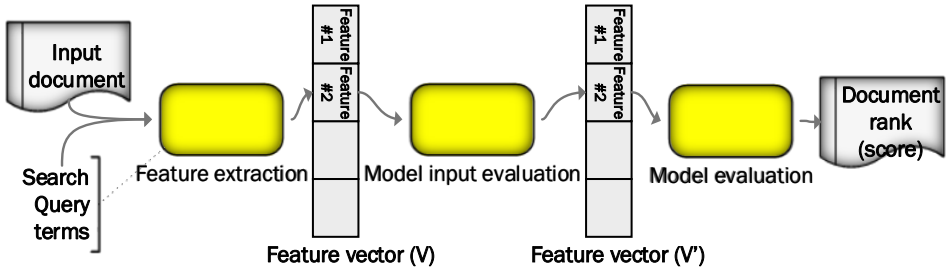

Figure 2.4: Document ranking steps. The yellow boxes are the GPU modules.

The second step is the document ranking step that utilizes an expensive second-stage ranker. This step works on a smaller number of documents per query. The second-stage expensive ranking phase extracts more documents features (static and dynamic features) and uses them to rank these documents, sorts them by that rank and returns top N documents to the aggregator.

Figure 2.4 shows the document ranking steps. It shows data flow and modules of the first or second stage rankers. The difference between the first stage or the second stage rankers is their complexity. A ranker's complexity is measured by the number of features that it needs to extract from the document. Complexity is also measured by the size of the ranker's machine learning model. Ranking one document involves three steps:

First, document feature extraction (FE). The input to this module is the intersection on the search query terms and the document that needs to be ranked. The output of this stage is a feature vector. The feature vector is a vector that contains all the needed document features values to evaluate the machine learning model that we will describe in model evaluation below.

The second step is ranking model inputs transformations (MIT). This step transforms the feature vector; that is generated from FE step. These transformations are one-to-one or many-to-one, which means one or more features from the original feature vector V are involved in calculating one feature in the transformed feature vector V'.

The third step is the ranking model evaluation (ME). This step takes the transformed feature vector V' and evaluates a machine learning model, The output of this step is one final number that represents the document rank or score. In this context, a document with a higher score means a better result for the user because it is more relevant to their search query. As part

of this thesis, feature extraction, model input transformation and ranking model evaluation were ported to the GPU. They are described in the next section.

Feature Extraction FE

In contemporary web search engines, there are many types of features that get extracted from each document that get ranked in the index.

First, "static features" are assigned by the crawler, and they are stored in the document metadata. The static document features are fixed per document, and they do not depend on the query. Using these static features for ranking requires copying their values from the document data to the feature vector only as no processing or calculations are required.

Second, "dynamic features" depend on the query terms. For a given document, a feature value differs for different query terms. Hence, dynamic feature extraction required online processing. Some examples of these dynamic features are the number of query word occurrences in the document.

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})}, \tag{2.1}$$

Finally, "Dependent features" which are a combination of other static and dynamic features, e.g., BM25F. BM25 feature is a document ranking feature that uses the number of query terms occurrences in the document' stream and the size of the document' stream. BM25 uses these two features to calculate the adjusted term frequency for each word and the inverse document frequency (IDF). Then it applies a formula to calculate the BM25 feature value. Equation (2.1) shows the BM25 formula. These dependent features have to be evaluated after the dynamic feature evaluation.

There are hundreds of different dynamic features that are involved in ranking each document.The dynamic feature extraction logic is represented by a finite state machine FSM.

Figure 2.5 describe the general flow of any feature based on the input document.

The document is feed to the FSM. The FSM extracts the document header then the stream header and finally the phrase and words. Most

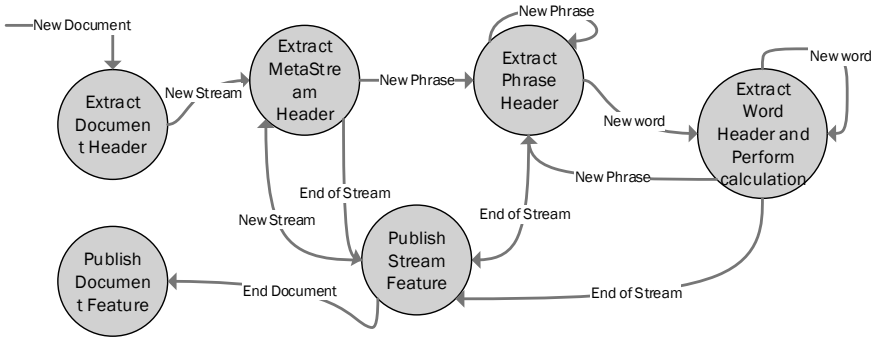

Figure 2.5: Dynamic Feature Extraction FSM

of the computation happens in the extract word state. The computation aggregation and feature value publishing happen in the following states: publish stream feature and publish document feature states. Each FSM can publish multiple different features. For example, the "Word Occurrence" FSM publishes the number of occurrences feature as well as the first and last occurrence features. The size of the feature output is variable, some features output a scalar value and other features output one and two-dimensional arrays.

Section 3.3 describes our GPU implementation of the feature extraction on the GPU.

Model Inputs Transformation MIT

Model inputs transformation is a transformation on the features vector from V to V'. The number of elements in the transformed feature vector V' differs from the number of items in the original feature vector V, which has implications for implementation on the GPU as described later. These transformations are categorized into two types: single feature transformation and multiple features transformations.

Single feature transformation allows the ranker designer to scale or clamp the feature input value. The memory access pattern, in this case, is regular where each V[i] is transformed into V'[i].

Examples of these transforms:

- **Linear:** The equation for a linear transformation is:

$$\text{output} = \text{input} \times \text{slope} + \text{intercept} \tag{2.2}$$

- **Log Linear:** The equation for a log linear transformation is:

$$\text{output} = \log(\text{input} + 1) \times \text{slope} + \text{intercept} \tag{2.3}$$

- **Bucket:** The equation for a bucket transformation is: output = 1 iff LowerLimit $\leq$ input $<$ UpperLimit output = 0 otherwise

$$\text{output} = \begin{cases} 1\text{iff LowerLimit ¡ input ¡= UpperLimit} \\ 0\text{otherwise} \end{cases} \tag{2.4}$$

Multiple features transformations allow the ranker designer to combine multiple features together. Examples of these transforms:

- **Mathematical Expressions (FreeForm):** The equation for the mathematical expressions transformation is:

$$\text{output} = \text{Function(input1, input2, ... inputn)} \tag{2.5}$$

- **Decision Trees:**

  Decision trees are unbalanced binary trees, and each tree has a topology, internal nodes with thresholds, and leaf nodes with output values. Decision trees models are trained offline and are ready to be used. The inputs to the decision tree are the extracted features, and the output is a single number that represents the tree evaluation score. Figure 2.6 shows a simple three layers decision tree with three internal (I1, I2, and I3) nodes and These leaf nodes (O1, O2, O3, O4). The real ranker decision tree may have hundreds to thousands of nodes. The decision tree evaluation starts at the root I1. Feature A is compared against threshold1. If feature A value is greater than threshold1 the next node is I2 else the next node is I3. The process keeps repeating until it reaches a leaf node. When this happens, the leaf node output value is returned as the result of this tree evaluation. As you can see the access pattern to the features in the feature vector and choosing the next tree nodes is irregular in the tree evaluation. Some methods can be applied to overcome the branch and memory divergence. These methods will be discussed in the GPU implementation.

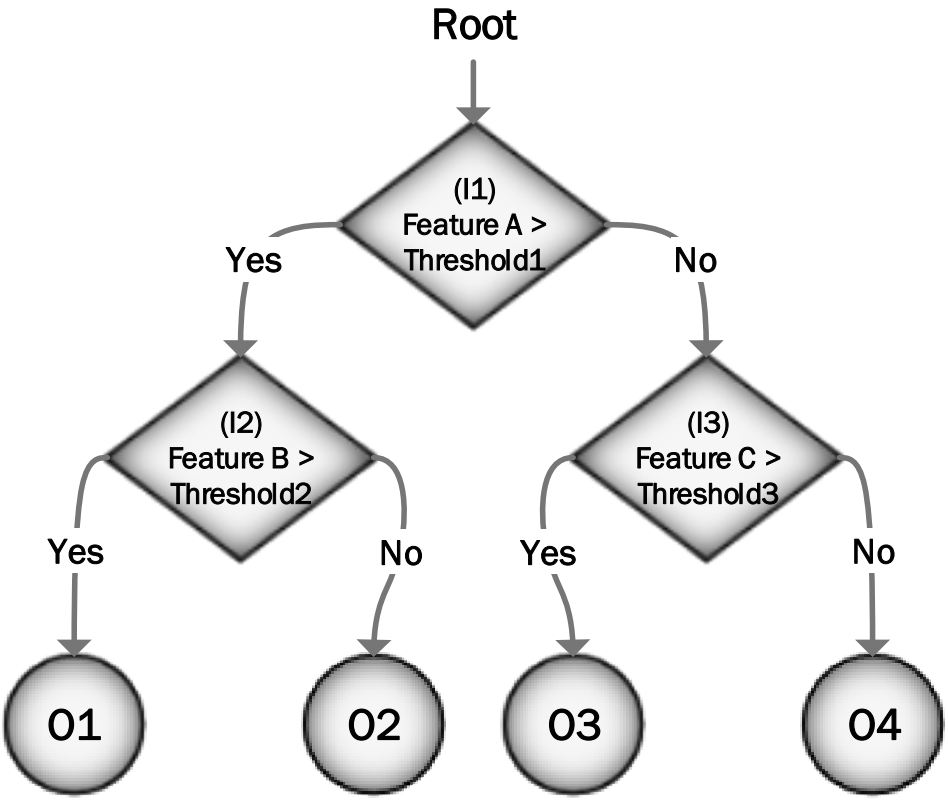

Figure 2.6: Simple three Layers Decision Tree.

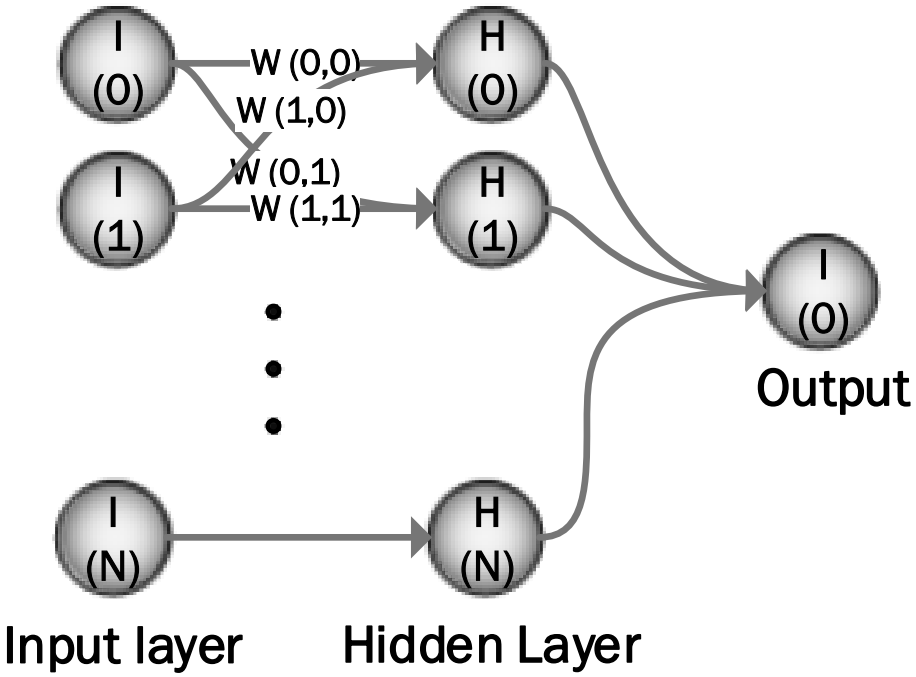

Figure 2.7: simple two layers neural net.

Model Evaluation ME

The machine learning models that are used in information retrieval are Neural Nets, Decision Trees, and a mix of multiple models which is called ensemble. Three steps are needed to use the machine learning model; First designing the model. Second training the model. Third is the execution or evaluation of the model. Designing and training the model happens offline by the relevance team, and the output of this training is the model description and the weights/thresholds. The trained models are typically considered the search engine's "secret sauce" (i.e., the source of competitive advantage). Model evaluation is the final step that calculates the document rank, and it is an online process. We chose neural nets that aggregate the features and decision trees outputs.

Figure 2.7 shows an example of a two layers neural net. The input is an array "I" of N elements. The first layer is Layer H. Layer H has N nodes. There are N $\times$ N weights between the inputs and the first layer. The values at any node H(x) is the sum of products of all the inputs.

$$H(i) = \sum_{j=0}^{N} I(j) * W(j,i) \tag{2.6}$$

Each layer is the sum of products of the previous layer. The final Layer is just one neuron.

2.2 GPUs

GPUs were introduced in 1999 for PC industry. They initially were developed to handle the computer graphics computation and rendering. The nature of the graphics algorithms is multiple independent operations with high parallelism. Each pixel that appears on the computer monitor requires transformation, lighting, triangle setup/clipping and rendering. Over the years, the computer graphics and games became sophisticated with more detailed. This advancement in computer graphics and games required faster processing speed that led to tremendous advancement in the GPUs. Currently, the two main vendors produce these graphics processors; NVidia and AMD.

Modern graphics processor units (GPUs) are organized around Multithreaded SIMD function units. Typically, an application starts on the CPU and launches a data parallel kernel onto the GPU. The kernel consists of a hierarchy scalar threads organized into small groups of 32-64 scalar threads,

called a warp in NVIDIA terminology, that execute in lock-step on the SIMD hardware. Divergent control flow within a warp is supported through various hardware mechanisms (e.g., prediction and related approaches). Tens of warps are grouped into a thread block and can communicate via a fast on-chip scratchpad memory (called "shared memory" in NVIDIA terminology). The warps in a thread block are active at the same time and fine-grain interleaved upon a single SIMD core. A small number of thread blocks from the same kernel may be scheduled at once on the same SIMD core. Typical GPUs contain tens of SIMD cores so that a hundred thread blocks and tens of thousands of scalar threads can be executing concurrently.

GPU vendors provided APIs to allow the programmers to write general purpose applications on them (GPGPU). There are few programming languages that can be used to write the general purpose applications that are targeted to run on the GPUs; examples are Nvidia CUDA [28] from NVidia and OpenCL [12] from AMD. Recently Microsoft announced the C++ Accelerated Massive Parallelism (C++ AMP) [22], which is a C++ language extension for developing applications on data-parallel hardware such as GPUs. Microsoft ships C++ AMP in Visual Studio 2012 IDE that contain the VC11 compiler. From Intel LLVM ShevlinPark project [37] "Our (subjective) experience: Writing data parallel code using C++ AMP is highly productive", we confirm this observation in our experience.

C++ AMP is an open specification that can be implemented by any vendor. Microsoft implements C++AMP on top of its DirectCompute and the High-Level Shader Language (HLSL). There are many features that make C++AMP excellent choice, and improve the programming productivity. Some of these features are:

- C++ functions can be marked with the keyword "restrict" to run on the CPU, the GPU, or both.
- The kernel starts at parallel_for_each library call that is very close to the parallel_for from the MS PPL library.
- The use of lambda expression and the function object to write the GPU kernel make it easier and faster to compose and reuse code.
- The use of the "tile_static" variables and the synchronization "tile_barrier::wait" alow for easy use of the shared memory on the GPU.
- C++ AMP comes with great libraries such as: "Concurrency::fast_math", "Concurrency::precise_math", and "Concurrency::amp_algorithms".

- Using “array” and ”array_view”objects to move the data between the CPU and the GPU easily and transparently to the developer. Also, no change needed to the code when running on integrated CPU-GPU on the same chip where data copy is not required.
- Using “accelerator” and ”accelerator_view”objects to decided where to execute the code make it easily to switch between the GPU hardware and the emulator.
- C++ AMP comes with excellent tools including the software emulator for debugging.

GPUs such NVIDIA's Kepler and Maxwell [3, 29] have been shown to increase throughput by up to an order-of-magnitude versus well optimized multicore CPU code [19]. This is possible because they dedicate more area to computation rather than complex instruction scheduling hardware and cache. Furthermore, the latest NVIDIA family of GPUs, Maxwell [3] is designed with power efficiency as its top priority. Maxwell architecture gives 2X performance per watt compared to its predecessor.

2.3 Summary

In this chapter, we provided the necessary background of the web search engines and the GPUs. We showed that the web search engine query processing requires three main computation steps: Feature extraction (FE), model input transformation (MIT), and model evaluation (ME). In following sections, we will also refer to the FE and MIT combined as Model input evaluation (MIE). The next chapter talks about our implementation, followed by the experimentation and the results chapters.

Chapter 3

Web Search on GPUs

As described in Chapter 2, the query processing workflow consists of two main phases; document selection and document ranking. Both phases involve the following steps; Reading the index (ISR), decompression, Feature extraction (FE), model input transformation (MIT), and model evaluation (ME). We will discuss the decision of offloading modules to the GPUs, and then we will describe the design and optimization for offloading each module. Next, we will talk about GPU memory management. Finally, we will talk about the deployment to production and failure handling.

3.1 Determining What to Offload to the GPU

We started by profiling the index server service that showed that a significant share of the CPU time is spent on Feature extraction, model input transformation, and model evaluation.

FE consists of multiple calculations. On average each feature performs one ALU operation per word in the request document. The CPU performs tens to hundreds of thousands of addition and multiplications per query to extract the document features. This amount of ALU operations makes FE ALU bound process. The MIT includes two operations per feature for the single feature transformation and multiple compare operations per multi-feature transformation. The ME consists of many addition and multiplications operations based on the model size. These steps are computation bound and not I/O bound, and they seem like a good candidate to be offloaded to an accelerator as other previous work tried accelerating similar modules [33].

We decided to offload the MIT and ME computation first because it has a high computation demand with less memory access. In this thesis, we also evaluate offloading FE.

For ranking model selection, we cover a large number of ranking models including neural nets with decision tree inputs. For feature extraction, we cover 125 features and some dependent features like the BM25 family of features.

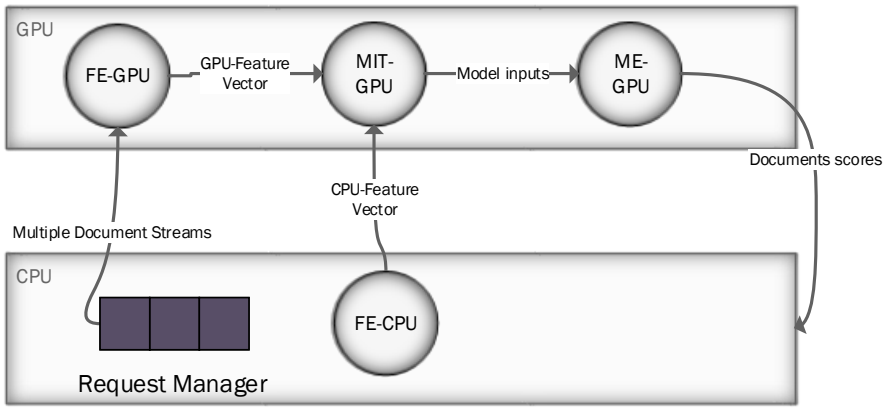

Figure 3.1: GPU implementation overview.

Any other features or models that are not currently supported in the GPU implementation are executed in the CPU, and the result is forwarded to the GPU when needed.

In the following sections, we describe the challenges of offloading the computation of each of these steps into the GPUs and our optimizations and implementation.

3.2 GPU Design Overview

Figure 3.1 shows the overview of our implementation. The CPU sends the requests to the GPU feature extraction FE-GPU. At the same time, the CPU does the CPU feature extraction for other features FE-CPU. Then the CPU sends the software evaluated features "CPU-FeatureVector" to the GPU model input transformation MIT-GPU. The GPU executes the FE-GPU then forward the resulting "GPU-FeatureVector" to the MIT-GPU. Next, the GPU executes the MIT-GPU and forwards the results to ME-GPU. Next, the GPU executes the ME-GPU. Finally, the CPU pulls the documents scores from the GPU memory.

```
DoId:123
    StreamId:1
        Phrase:1 Count:1 Length:10
            word:0 offset 10
            word:0 offset 12
            word:1 offset 13
... and so on for the other streams and phrases.
```

Figure 3.2: Example FE request.

3.3 Feature Extraction on GPU

Feature extraction (see section 2.1.4) on GPUs is challenging because the FE logic is represented by finite state machine FSM. It is hard to parallelize the FSMs execution because the execution pattern is serialized. Partitioning the FSMs input and creating a thread per input partition does not provide a performance benefit. This poor performance happens because of the inherent dependencies in execution. For example, the second thread start state is the first thread end state. The second thread has to wait for the first thread to finish that leads to serializing the work. We overcome some of these dependency issues by using the observation that the document is divided into streams and most of the features operate on each stream independently.

We will describe our FE-GPU request format then we will describe the FE optimizations that we implemented. Each document consists of multiple document streams. Moreover, each stream consists of multiple phrases. Also, each phrase includes multiple word hits. Word hits mean where did the user keywords appeared in the phrase. The query and document pair arrive at the Request Manager. The Request Manager open the ISR and creates the FE requests on the CPU. The FE request contains the hit vector which mean where do the query terms appear in the document.

An example of a document request is:

Query: “Lazy Dog” (Word:0 is “Lazy” and Word:1 is “Dog”) The GPU-FE request to extract the features for document identifier 123 is shown in Figure 3.2.

This request means document identifier is 123 which can be something like:

http://www.example.com/examplepage.html

StreamId:1 could be the body section of that web page.

Phrase:1 is one phrase in the document.

Figure 3.3: GPU FE serial request format.

Word:0 "Lazy" appeared at location 10 in the first phrase.
Word:0 "Lazy" appeared at location 12 in the first phrase.
Word:1 "Dog" appeared at location 13 in the first phrase.
and so on ...

The first, un-optimized serial approach is to send one document at a time to the GPU. The data structure for this single document is a one-dimensional array. The GPU has a single thread that loops over the input document to evaluate each feature. This approach is the GPU baseline. There is two type of data that we need to move to the GPU memory: The ranking model specific data and the query/document specific data. The ranking model specific data for FE can be moved once per ranking model GPU loading. More on that later in the section that talks about the GPU models memory management. The query/document specific data have to be moved per query. In case of FE, the data sent to the GPU is the query/document request.

The factors that affect the execution time for one document are:

- Time to copy the document
- Time to launch the kernel
- The execution time for each feature multiplied by the size of the document.

The request format for the GPU baseline is described in figure 3.3. We send the document header followed by the number of streams. Next is the stream header followed by the number of phrases. Next is the phrase header followed by the number of words. Finally the list of the words.

The next optimizations exploit a higher degree of parallelism; they change the memory access pattern of ranking one request, and they combine multiple requests to gain more parallelism.

The first optimization that is specific to the ranking problem leverages the observation that a single document contains multiple streams, and each

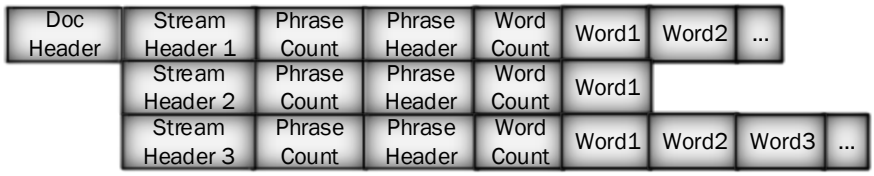

Figure 3.4: GPU FE request format-parallel streams.

stream is independent with respect to many features. Figure 3.4 show the request format for this optimization. In this optimization, we parallelize the document based on the number of streams that it contains. When we execute the feature extraction on the GPU, we launch one thread per stream. Some streams are much larger than other streams. In this approach, the latency of the request depends on the largest stream. One way to increase the GPU utilization is to combine two or more small streams together when their total size is smaller than the largest stream.

The second optimization is executing each feature in a different warp to allow multiple features to execute at the same time. This method can reduce the GPU utilization because some warps have unused threads, but it reduce the request latency. The warp size is 32, and we make each 32 threads handle executing one feature. Furthermore, by combining the first and the second optimization, each 32 threads are executing multiple streams for one different feature.

The obvious final optimization is batching multiple request to the GPU that contains a group of documents to be ranked together. This approach required the introduction of a request queue. Multiple software worker threads fill the queue, and one software thread reads the queue periodically (based on a threshold) batch the requests and send them to the GPU. This approach allows us to fall back to CPU only execution if the number of requests per period is not enough for the GPU to get a benefit. Also the fall back to CPU can be used in the case of GPU hardware or software failure, and we will discuss this in the failure handling section. In the results chapter, we will show the number of documents that the GPU implementation is better than the CPU. This number depends on the ranker complexity as we will explain there.

3.3.1 Dependent Feature Extraction

Some features depend on other features to be calculated first. An example of such a feature, we implemented the well known BM25F [35] ranking function; BM25F features are dynamic document ranking functions; they rank the document relative to the query terms. The simple form of this function uses the number of query term occurrences in the document's stream (which is another feature) and the size of the document's stream to calculate the score. Our implementation of BM25F uses two-dimensional GPU array. The first dimension is the number of query words; the second dimension is the number of documents to be ranked. For each query word, we calculate the adjusted term frequency for that word and the inverse document frequency (IDF). Next step, we use the *Word0* thread (the thread that handles the computation of first query word) for each document to aggregate the data, sum the IDF and calculate the BM25 initial sum. Finally, we scale the BM25F and calculate the normalized version too.

Once the document features are generated in a "feature vector" form, the ranker performs a transformation on this feature vector, called the model input transformation.

3.4 Model Input Transformation

After evaluating FE-GPU, FE-CPU and copy CPU feature vector to the GPU memory, the resulting GPU feature vector, and the copied CPU feature vector are combined in the GPU memory. The result is the document feature vector "V" that is the input to the second step; Model input transformation.

In this phase, a feature vector V is transformed to V'. The single and multiple feature transformations are challenging to offload to the GPU because of their memory and control divergence. The memory access patterns are irregular specially for the multiple feature transformations because each element in V' is a different transformation and gets calculated using multiple distanced values from the original feature vector V. Next, we will describe our data structure design and the transformation implementation and our proposed optimizations.

The data structure of the MIT contains the transformation type, the transformation calculation parameters, and the index of the feature in the feature vector. The data structure is described below:

The transform types are shown in code 3.5,. These transformations are the single feature transformation: Linear, LogLinear, and Bucket. Moreover, the multiple features transformations: DecisionTree. The dummy transfor-

```
enum GPUTransformType
{
    Dummy,
    Linear,
    LogLinear,
    Bucket,
    DecisionTree
};
```

Figure 3.5: Model Input Transformation Types.

```
struct GPUModelInputTransformation
{
    GPUTransformType m_transform;
    unsigned int m_featureIndex;
    float m_param1;
    float m_param2;
};
```

Figure 3.6: Model input transformation structure.

mation is used for padding in the warp optimization that will be described later.

For each transformation in the final feature vector, there is one structure that represent it as in code 3.6. m_transform specifies the transformation type. m_featureIndex specifies the index of the feature that needs to be transformed.

The transformation parameters are param1 and param2. These two parameters have a different interpretation based on the transform type. Ta-

Transformation Type	Param1	Param2	FeatureIndex
Dummy	N/A	N/A	N/A
Linear	Slope	Intercept	Input feature' index
LogLinear	Slope	Intercept	Input feature' index
Bucket	LowerLimit	UpperLimit	Input feature' index
DecisionTree	N/A	N/A	Index of tree toplogy array

Table 3.1: MIT param1, param2 and FeatureIndex meaning per transformation type.

```
struct GPUTreeNode
{
    unsigned int m_treeFeatureIndex;
    unsigned int m_treeNodeLTE;
    float m_treeNodeThreshold;
};
```

Figure 3.7: TreeNode structure.

```
for (unsigned int i = 0; i < numberOfInputs; ++i)
{
  float featureValue=p_input[m_GPUMIT[i].m_featureIndex];
  switch (m_GPUMIT[i].m_transformationType)
  {
    case TransformationType::Linear:
        OutputFeatureVector[i] = (featureValue *
          m_neuralInputParameters[i].m_param1) +
          m_neuralInputParameters[i].m_param2;
    break;

    case TransformationType::LogLinear:
        ...
  }
}
```

Figure 3.8: MIT unoptimized.

ble 3.1 shows the meaning of them at various transformation types.

Decision trees transformations are handled differently. For decision trees, the FeatureIndex is a pointer to an offset in the tree topology structure shown in figure 3.7. Each node in the tree has one GPUTreeNode struct. In this struct, m_treeNodeThreshold is the node threshold. m_treeFeatureIndex is the index of the feature that will be compared against the m_treeNodeThreshold. m_treeNodeLTE is the index of the left child node, and the index of the right child node is m_treeNodeLTE + 1;

The simple GPU implementation is a serial implementation. For each element in the vector V', it calculates the transformation. The code in 3.8 shows the initial implementation.

The first optimization is to launch a thread per transformation.

The second optimization is to rewrite the ranker description file to re-ordering the transformation to group the similar transformations together.

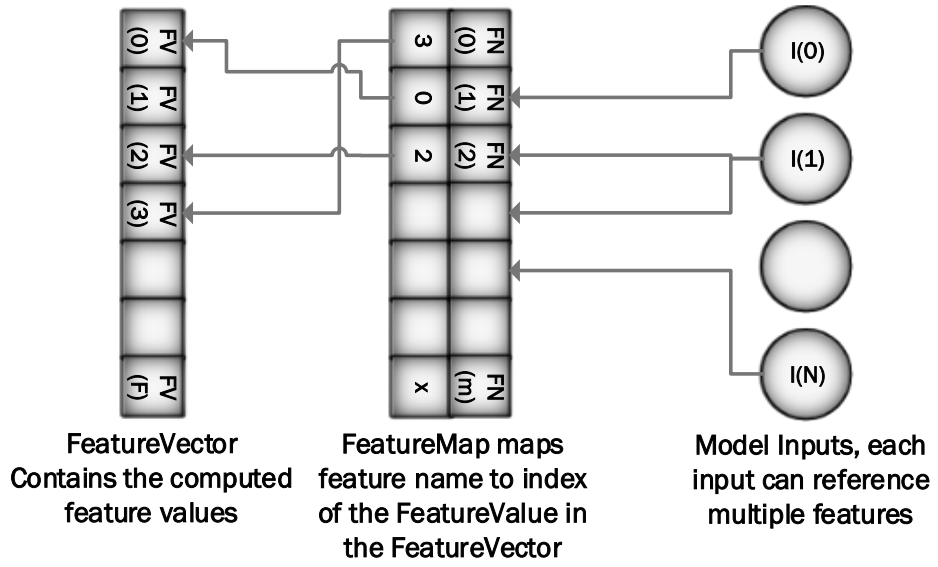

Figure 3.9: Model inputs reference features in feature vector using the feature map.

The ranker rewriter analyzes the input ranker description file and builds lists of transformation groups. It uses that groups of transformations to rewrite the file, so the similar transformation types appear adjacent to each other. It also has to swap the model weights to reflect the inputs reordering.

The third optimization is to add padding using the TransformationType::Dummy to make each warp of 32 threads execute one type of transformation. This allows multiple transformations to be executed at the same time.

The fourth optimization is to make each transformation group of threads from V' use features close together in the input feature vector V. To achieve high memory access bandwidth, the data accessed by different threads in a warp should be nearly adjacent so memory coalescing can be more effective. We achieve this by modifying the feature map. The feature map is the object that assigns the feature indexes in the feature vector. It works as follows: each model input references one or more document feature(s) by asking the feature map about their index in the feature vector. Figure 3.9 shows model inputs I_i accessing some feature values FV_{ii} using the "feature map" name to index mapping from FN_x to FI_y. The feature map is important in achieving high memory access bandwidth because it utilizes the spatial

```
parallel_for_each(gpuAccelerator_view, GPUEvaluatedInputs.extent,
[&](index<2> idx) restrict(amp)
{
  int documentNumber = idx[0];
  int inputNumber = idx[1];
  float slope = GPUMIT(inputNumber).m_param1;
  float intercept = GPUMIT(inputNumber).m_param2;
  GPUTransformType transformationType =
        GPUMIT(inputNumber).m_transform;
  UINT32 featureIndex =
        GPUMIT(inputNumber).m_featureIndex;

  if (transformationType == TransformationType::Linear)
  {
    float featureValue = (float)GPUFeatureVector(documentNumber, featureIndex);
    GPUEvaluatedInputs(documentNumber, inputNumber) =
        (featureValue * slope) + intercept;
  }
  ...
}
```

Figure 3.10: MIT optimized.

locality, e.g., adjacent model input threads access adjacent feature values in the feature vector.

The last optimization is to batch a group of documents together. Our implementation utilizes a thread per final transformation. The code in figure 3.10 shows the final optimized code.

The data copy from the CPU to the GPU is critical for performance. We experimented with different precisions (single point vs. double point) and achieved a very close numbers compared to the CPU-only ranking and better than Fixed point computations. The data copy from the main memory to the CPU includes:

1. Moving the CPU feature vectors to GPU memory:

   The size of the model inputs is N, the feature vector size for each document is F, where N $\neq$ F because multiple inputs can reference the same document feature, and some inputs can reference multiple document features. When GPU ranker evaluates D documents, we have to move D * F * 4 bytes (feature value) from main memory to the GPU memory. This data moving happens each time a batch of

documents needs to be ranked.

2. Moving the ranker input definitions to the GPU memory:

   The layout of the in-memory data structure that represents the ranker is critical. This is because it changes the memory access pattern when evaluating the rank of each document, we experimented with a different layout, e.g., an array of structures vs. a structure of arrays [11]. For each model input, we have to copy the model input transformation for each model input. We use a common structure shown in figure 3.6 for the single and multiple features model inputs but with different fields meaning. Alternatively, we tried the structure that hold arrays of m_transform, m_featureIndex, m_param1, and m_param2.

   All the trees data are moved to the GPU in a vector of GPUTreeNode shown in figure 3.7. For each tree input, the `m_featureIndex` in the GPUModelInput structure carries the offset of the root tree node in the GPUTreeNode vector. The field `m_treeFeatureIndex`, on line 3, is the feature index for the feature that this tree node needs to check against its threshold.

   The field `m_treeNodeLTE` on line 4 represents the tree topology by giving the index of the left child node. When a tree node check the feature against its own threshold, if feature value is less than or equal the threshold, the next tree node index in the GPUTreeNode vector is `m_treeNodeLTE` + offset, otherwise: `m_treeNodeLTE`+ 1 + offset.

   The field `m_treeNodeThreshold`, on line 5, carries the tree node threshold, if the `m_treeFeatureIndex` is -1 then this node is a leaf node and we use the `m_treeNodeThreshold` to store the leaf node output value.

   The model input definition data is moved to the GPU once per model load.

3.5 Model Evaluation

As we described before, document ranking uses two well-known machine learning algorithms; neural network and decision trees. The ranking model combines the algorithms together. In the following section, we will focus on neural networks that have some of their neural input made as decision trees.

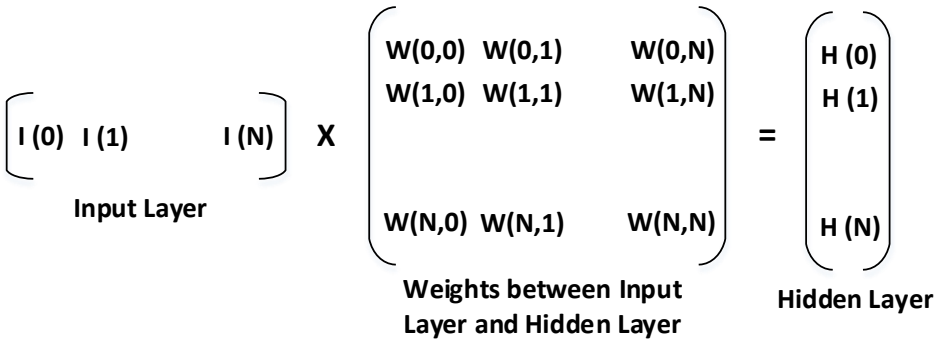

Figure 3.11: Modeling neural network evaluation as matrix multiplication for single document.

3.5.1 Neural Network Evaluation

For the neural network implementation, most of the work on neural network evaluation on the GPU involves translating the neural network to matrix multiplication as in [31]. Matrix multiplication maps very well in GPUs.

The first approach is to rank each document individually. Using the neural network from Figure 2.7, Figure 3.11 shows how to rank the single document. The input is the array of size N neural inputs I(0) to I(N). We multiply this by a matrix of size NxN, which represents the weights between the input layer and the second layer (or the hidden layer). The result array of N represents the output at the first layer of the neural network. This process carries on for each layer of the neural network until we reach the last layer. For example, for a neural net with three layers this multiplication have to be done three times. Depending on the size of each layer of the neural net, this implementation can be very expensive compared to the CPU version. The larger the neural net layer size, the more efficient is the GPU implementation. To get the benefit of the massive parallelism in the GPU hardware, we rank multiple documents at the same time.

We expand the first approach to evaluating the neural network model for multiple documents in parallel. Figure 3.12 shows evaluating D documents on the neural network in Figure 2.7. The input is a matrix of size DxN neural inputs, and the output is a matrix of size DxN that represent the output of the first layer. After evaluating the last layer of the neural network, the result is an array of size D that contains the ranks of the documents. This array get sent back to the CPU. The code in figure 3.13 shows the neural

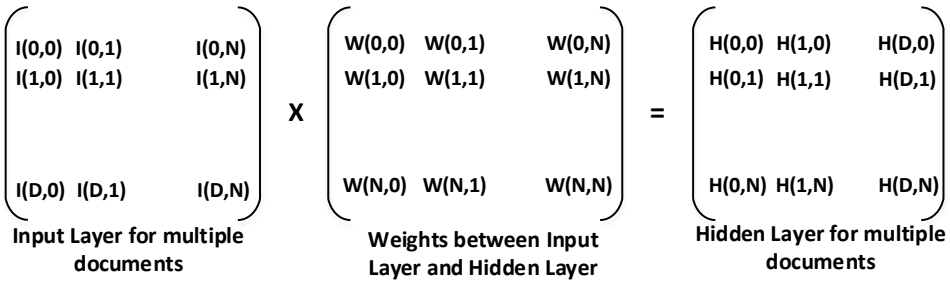

Figure 3.12: Modeling neural network evaluation as matrix multiplication for multiple documents.

```
parallel_for_each(gpuAccelerator_view, GPUresultGPU.extent,
[&, numberOfInputs](index<2> idx) restrict(amp)
{
    int documentNumber = idx[0];
    int nodeNumberinLayer0 = idx[1];

    float sum = GPUWeights(nodeNumberinLayer0, 0);

    for (unsigned int i = 0; i<numberOfInputs; ++i)
    {
        sum += GPUEvaluatedInputs(documentNumber, i) *
               GPUWeights(nodeNumberinLayer0, i+1);
    }

    GPUresultGPU(idx) = sum;
});
```

Figure 3.13: Neural Net evaluation of one layer of the for multiple documents.

net evaluation of one layer of the for multiple documents.

We use the optimized version of C++AMP matrix multiplication [11] which utilize the GPU shared memory for fast accessing the repeated accessed data.

3.6 GPU Models Memory Management

As described before the data that we move to the GPU are two types:

1. Model-dependent data. This data get loaded once per model load, and it is reused for each query. The model-dependent data are the MIT data including the trees topology and the neural network description data.

2. Model-independent data which is the document/query data. This data is per query, and it has to be sent to the GPU for each query.

The neural network description data is the weight matrices. There is one weight matrix per neural network layer of the model. These matrices need to be moved to the GPU before the evaluation of the model on the documents by the CPU. A part of the GPU global memory is assigned to hold the models data. Multiple models can be loaded in the GPU memory at the same time. The CPU keeps track of the loaded models in the GPU memory and pass the pointer to the GPU kernel to evaluate the right model. If the required model is not loaded in the GPU memory, the CPU copy the data to the GPU with the first query that required the "unloaded" model. If the model data is not loaded in the CPU memory, the CPU has to load the data from a ranker's description file that exists on the disk. The CPU loads these weight matrices and sends them to the GPU to be cached in the GPU memory. The weight matrixes stay in the GPU memory until it is evicted/replaced by the CPU. The module that manages the GPU memory is called the GPU ranker cache manager. The GPU ranker cache manager uses LRU to manage the GPU memory.

3.7 Software Deployment and Failure Handeling

The GPU accelerated index serving software deployment is not much different from the non-accelerated one. The introduction of a new hardware (The GPU) in each server requires driver installation and upgrade which can be handled like the operating system (OS) update.

For the failure handling. Each cluster of ISNs should contain some machines with GPUs dedicated to replacing failed machines, these machines are called spare machines. In case of software failure, the same existing failure handling mechanism should kick in to restart or reimage the machine. In the event of a hardware failure (GPU), a decision has to be made. One way to handle that is to fall back to the CPU software only. Another way is to declare the machine as failed machine and use a machine from the spare pool.

For production, some infrastructure has to be made. This infrastructure includes perf counters, health monitors, alerts, and reports for the GPU.

3.8 Summary

In this chapter, we described the challenges of accelerating the query processing steps (FE, MIT, and ME) on the GPU. Then we described our design and how we overcome some of these challenges. We described in detail our implementation of the FE, MIT, and the ME steps, and we gave a list of the optimizations that we applied on each of them. Some of these optimizations are specific to the index serving problem. While other optimizations are general and can be used in other GPU acceleration applications. We concluded this chapter by talking about handling multiple rankers in the GPU memory and talking about the software deployment and the failure handling.

The next chapter talks about our experimentation methodology, followed by our experimentation results.

Chapter 4

Experimental Methodology

We implemented the GPU acceleration in a commercial web search engine. All the experiments that we ran on this search engine use real production data. We used the current system as a baseline for the CPU-only implementation. The baseline CPU implementation is highly optimized production implementation. We compared the baseline to our CPU-GPU implementation.

The execution time is measured using the Microsoft Windows high-resolution performance timer API [26]. The GPU implementation is written using Microsoft C++ AMP, VC12 C++ compiler and Visual Studio 2013. See Table 4.1 for details of the hardware used during our evaluation.

We run a production query log that we obtained from a production machine. We run hundred thousand queries/documents per model.

For performance/execution time experiments we run the GPU kernel one time first to allow the JIT to optimize the kernel, this happens only once per ranking service start. After the JIT optimizes the kernel, we run each experiment 10 times, and we measure the data copy for GPU and execution time of the CPU and the GPU. We used the Microsoft Windows high-resolution performance timer API [23] to measure the copy and execution time. We report the average of these runs.

For analysis experiments, we use Visual Studio "Concurrency Visualizer" profiler [10]. To read the hardware counters we use the NVIDIA PerfKit 3.1 [30]. We instrument the driver and use the C++ AMP DirectX interop to obtain the ID3D11Device object (which is a device representation in DirectX). We feed the ID3D11Device to the NVIDIA PerfKit to read the device counters by injecting code to read the hardware and the driver counters.

For the power measurements, we use the GPU-Z tool [39] to read the hardware sensors and log the power data to a file. The sample rate is 1 per second. For more accurate power measurements, we use the "Watts Up Pro" hardware tool [7] to measure the full system power. The sample rate is 1 per second. For the power experiments, we run a constant load of 10000 documents per query for 20 seconds, and we record the average power

CPU:	
Name	Intel Xeon W3690 (Westmere-WS) @ 3.47GHz
Technology	32 nm
TDP Limit	130 Watts
L1 Data cache	6 x 32 KBytes, 8-way set associative, 64-byte line size
L1 Instruction cache	6 x 32 KBytes, 4-way set associative, 64-byte line size
L2 cache	6 x 256 KBytes, 8-way set associative, 64-byte line size
L3 cache	12 MBytes, 16-way set associative, 64-byte line size
Memory Type	DDR3 Triple Channels
Memory Size	24 GBytes
Chipset:	
Northbridge	Intel X58 rev. 13
Southbridge	Intel 82801JR (ICH10R) rev. 00
PCI-E Max Link Width	x16
GPU Kepler:	
Name	NVIDIA GeForce GTX 660 Ti (GK104) @ 980MHz
Technology	28 nm
CUDA Cores	640
Memory size	2 GB @ 6.0Gbps
Memory Interface	GDDR5
Memory Interface Width	192-bit
Memory Bandwidth	144.2(GB/sec)
Maximum TDP	150 Watts
GPU Maxwell:	
Name	NVIDIA GeForce GTX 750 Ti
Technology	28 nm
CUDA Cores	1344
Memory size	2 GB @ 6.0Gbps
Memory Interface	GDDR5
Memory Interface Width	192-bit
Memory Bandwidth	86.4(GB/sec)
Maximum TDP	60 Watts

Table 4.1: Hardware specs.

Ranker Name	R-S-BLG	R-M-BGD	R-L-D
Number of Input	125	515	1250
Number of Layers	1	1	2
Inputs Types:			
Bucket (B)	54	10	0
Linear (L)	2	0	0
Log Linear (G)	69	5	0
Decision Tree (D)	0	500	1250

Table 4.2: R-S-BLG, R-M-BGD, R-L-D ranker details.

consumption.

In the first power experiment, we measure the system idle power consumption with no GPU (around 70 watts). Then we run the CPU implementation only for the three ranker types (described below), and we record the system power consumption. Finally, we subtract experiment power from the idle system power to get the CPU-only power.

In second power experiment, we add the GPU to the system, and we measure the full system power under constant load for the three rankers and we subtract that from the system idle power with no GPU.

For each experiment, we compare the document's rank output of both the CPU and the CPU-GPU implementation for correctness testing.

The index server has a ranker store that contain production and experimentation rankers. By analyzing and parsing a large number of both production and experimentation rankers, we decided to include three types of rankers based on their size. The selected three rankers represent a larger number of other slightly different rankers. The actual production rankers change over time, and this is a snapshot at the time of the experimentation. We will refer to them as R-S-BLG, R-M-BGD, and R-L-D.

Table 4.2 shows these rankers' parameters. R-S-BLG is simple ranker, then R-M-BGD is an intermediate complexity ranker. Finally, R-L-D is bigger and more complex ranker.

R-S-BLG represents the fast and small size ranker with 125 model inputs; these 125 model inputs have 125 single feature model input transformation without any decision trees. these single feature transformations are B: bucket, L: Linear, and G: Log-linear.

R-M-BGD is an intermediate size ranker with 515 model inputs. 15 model input are the simple single feature model input transformation. B:

bucket and G: Log-linear. It also has 500 decision trees (D: Decision Tree). Each one of these decision trees has N requested features.

R-L-D is a large and complex ranker with 1250 model inputs. These 1250 inputs are multiple features decision trees transformations (D: Decision Tree) without any single feature model input transformations.

For each ranker, we run the single thread CPU, parallel CPU, Kepler GPU, and Maxwell GPU. The single thread CPU is the current search engine implementation. In this implementation, each query runs on a single thread, and multiple queries can run on parallel. We use 12 threads, and each thread ranks all the documents in serial per query. When a thread finishes ranking all the documents in a query, it gets a new query from queries queue (this is called Work Stealing Queue). The parallel CPU is a multithreaded implementation, and it is not implemented in the production code for many reasons out of the scope of this thesis. We implemented it to compare with the GPU implementation. In this implementation, we use 12 threads per query. We pick query from the queries queue and run the 12 threads to rank the documents in parallel for the same query. This effect the query latency but it doesn't effect the CPU throughput. The GPU-Kepler and GPU-Maxwell are the same implementations on different GPU hardware. This implementation is described in detail in chapter 3. Please see the result section for comparisons.

Chapter 5

Experimental Results

Figure 5.1 shows the execution time for the simple R-S-BLG. We observe the following. First, ranking one document is faster on the CPU. The GPU implementation has overhead which includes; the time it takes to copy data to the GPU, queue a GPU kernel, run the kernel, and copy back the data, which is much more overhead than simply running on the CPU. The tradeoff changes when ranking 1000 documents where GPU total execution time becomes better than the CPU. In the a real system, the selection phase that uses small rankers like the R-S-BLG typically ranks more than 1000 documents. For 1000 documents and more we see the benefit of the GPU accelerated ranker.

Figure 5.2 shows details of the GPU execution time. In this figure, execution time is divided into five categories. The first three categories from the bottom are for the data copy from and to the GPU. Copy-FV-MI means copying the feature vector and the model inputs transformations. Copy-M means copying the model data. Copy-R means copying the resulting documents rank score back to the CPU memory. GPU-IE means the model input evaluation and GPU-ME means the model evaluation itself. From Figure 5.2 we can see that data copy can take up to 68% of the GPU execution time. This copying time can be avoided by using integrated GPU like AMD's Fusion APUs [2] the evaluation of which is beyond the scope of this work. Also, we notice that the model description copy "Copy-R" can be negligible at a higher documents number that suggested that we do not need to cache them as we do in the CPU. A fast ranker like this one is used already to rank a high number of documents, and GPU latency saving could be used to traverse more index documents, which leads to serving bigger index.

Figure 5.3 shows the execution time for the intermediate complexity R-M-BGD. We observe the following. First, it follows the same pattern as R-S-BLG, but the switching point where the GPU is performing better than the CPU happens at a smaller numbers of documents. At 500 documents, the GPU latency starts to be better than the CPU. This shows that when the ranker complexity increases, CPU performs lower and the GPU run

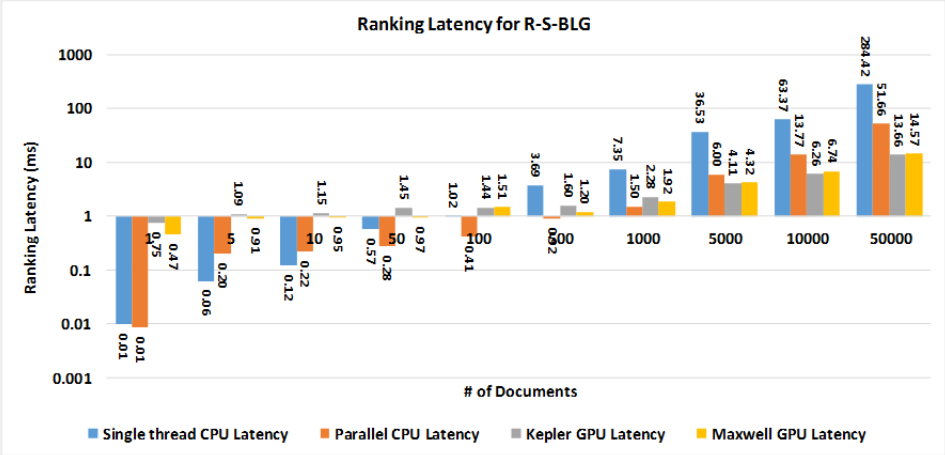

Figure 5.1: R-S-BLG execution time on the CPU and the GPU.

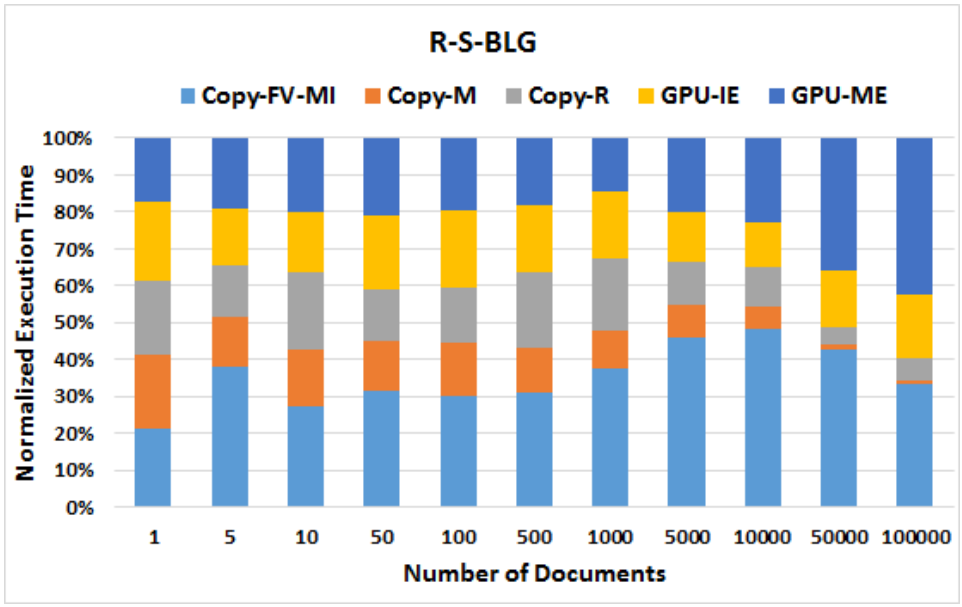

Figure 5.2: Details of R-S-BLG normalized execution on the GPU.

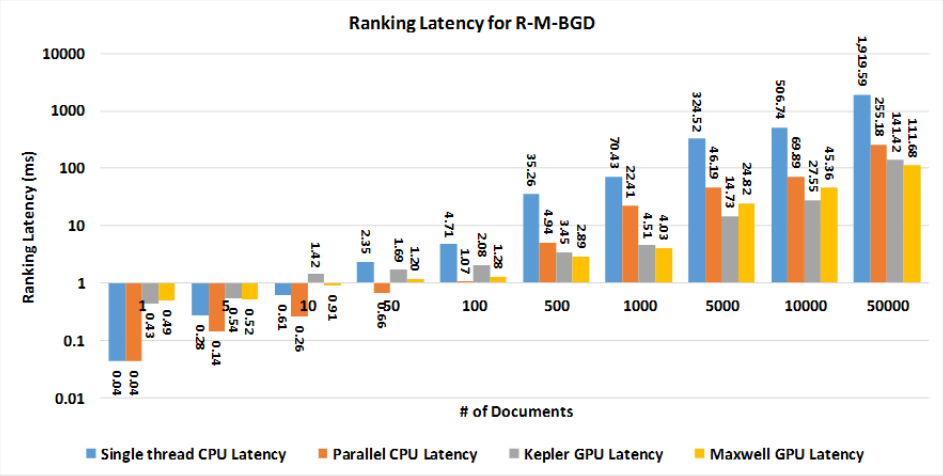

Figure 5.3: R-M-BGD execution time on the CPU and the GPU.

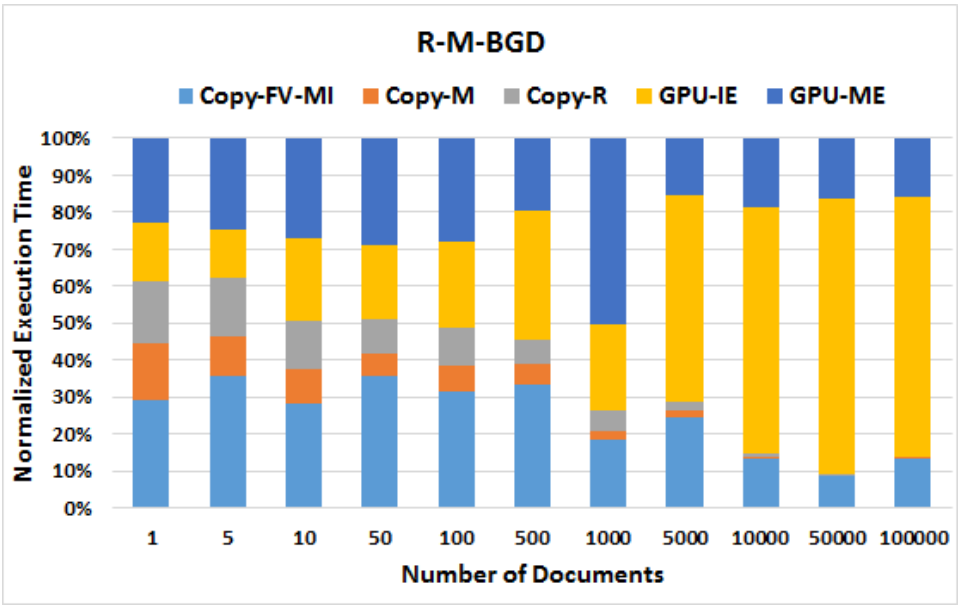

Figure 5.4: Details of R-M-BGD normalized execution on the GPU.

overhead effect decreases at a smaller number of documents. This result suggests that the GPU is more beneficial when the ranking models are more complex.

From 5.4 we can see that the time taken for data copy percentage is lower, and the kernel execution is higher. This happens because decision trees evaluation requires more computation than simple single feature transformation. Furthermore, each tree introduces very different memory access pattern than the single feature model input. In the single feature model input evaluation, each input transformation thread required single feature from the feature vector and these features are adjacent. On the other hand, the single tree requires multiple features that are multiple places in the feature vector and not necessarily adjacent. These reasons cause the GPU-IE component to be dominant. The model in R-M-BGD is still simple that is why GPU-ME is smaller.

Figure 5.5 shows the execution time for the complex R-L-D. It follows the same pattern as R-S-BLG and R-M-BGD, but again the switch point happens at an even smaller numbers of documents. At 50 documents, the GPU starts to achieve lower latency than the CPU. The data in figure 5.6 shows that trees are dominating the execution time "GPU-IE". We reach 19× improvement in latency for complete batch on GPU at 10,000 documents because GPU memory throughput is much higher than the CPU. Also, it important to observe that for the CPU version it was impossible to use such a complex model to rank more then 500 documents per query (70.61 ms). That is why such a ranker will be used in the "second-stage expensive ranker" phase. On the other hand, for the GPU implementation it can be used to rank 10000 documents. In fact, with the GPU implementation this could give the relevance developer and researchers an excellent tool to allow them to run more complex rankers on the "Fast first-pass ranking" in ranking phase.

5.1 Power Consumption

Using GPU-Z, the idle power consumption for Nvidia Geforce GTX 660TI (Kepler) is around 12.9% TDP (19.35W). And the idle power consumption for GTX 750 TI (Maxwell) is around 2.4%TDP (1.44W). The full system idle power is around 67W measured using "Watts up pro" Figure 5.7 shows the full system average power consumption for ranking 10000 documents for the three ranking model types. The CPU power consumption is worst for the R-S-BLG compared to the GPU. This happens because this Ranker has an

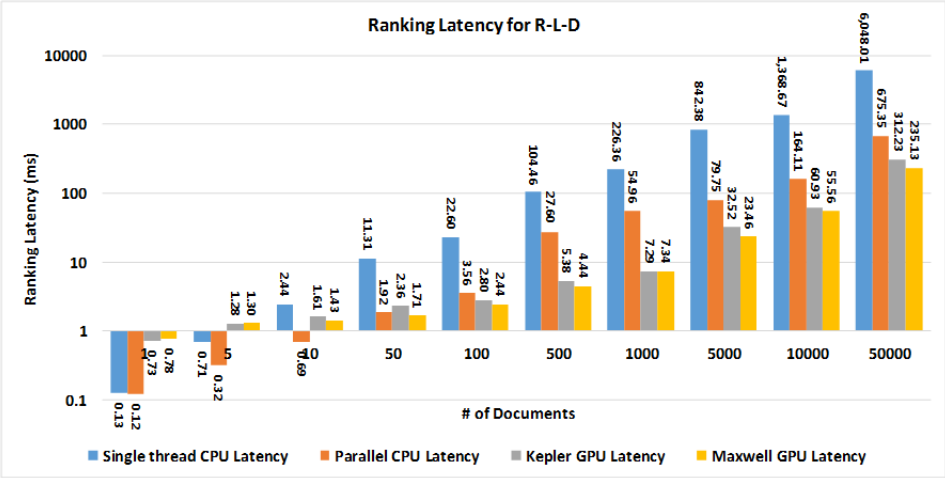

Figure 5.5: R-L-D execution time on the CPU and the GPU.

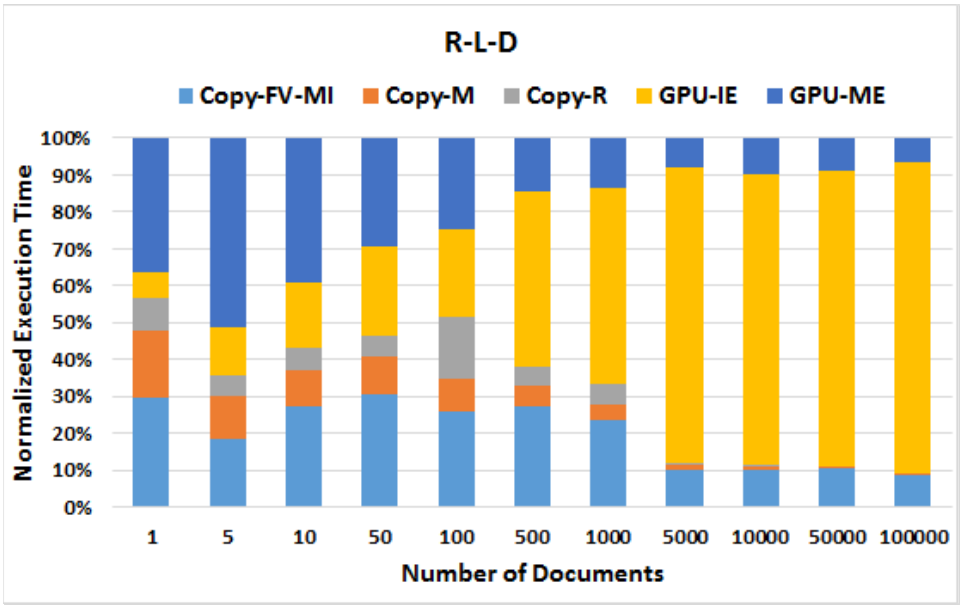

Figure 5.6: Details of R-L-D normalized execution on the GPU.

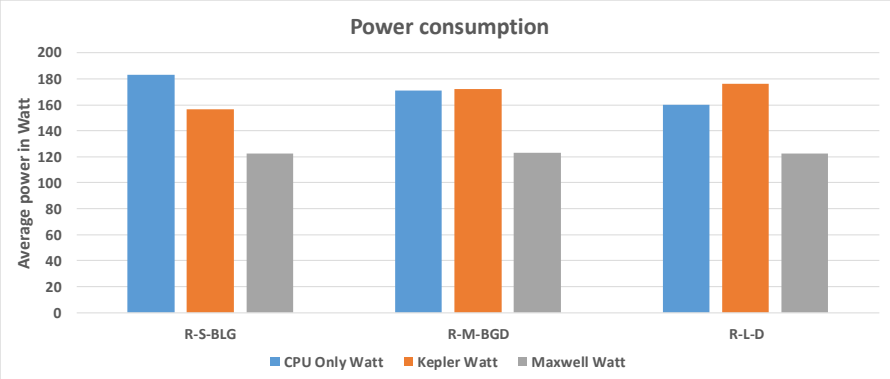

Figure 5.7: Average Power Consumption per Ranker Type

insignificant memory and control divergent, and the GPU implementation and execution is very efficient. The bigger rankers with more decision trees suffer from code diversion, and the CPU power consumption is better than Kepler GPU for R-L-G. The Maxwell GPU power consumption is better than CPU and Kepler CPU because it is optimized for power efficiency. This result shows the power efficiency gains that GPUs can bring to the data center workload such as the web search engine workload. Subtracting the system idle power consumption, the GPU can provide 2× power efficiency than the CPU at 10000 document per query for mid-size ranker like the R-M-BGD.

Chapter 6

Related Work

Some recent work examined a ways to parallelize the FSM [48]. Offline training for speculation is not efficient when the input data change. Prefix-sum parallelization uses many threads per data partition and starts from all the possible states [18, 25]. This approach has scalability issues. E.g., the number of useless computations increase linearly with the number of FSM states. Principled speculation [47] uses probabilistic "make-span" model (profit of speculation) based on offline training.

Some work is done to speed up the DNN for speech recognition using ARM and SEE [41, 42]. This work reduces the size of the model and converts the floating point to fixed point. While the speed up is great, the quality could be affected. Also, for search engines, the same approach could be used.

Reddi et al. [15] examined the use of power-efficient mobile cores in the search engines. While the measured power consumption was lower this affects the quality of search.

Ding et al. [8] studied using GPU in the web search engine. First, it focuses on the index compression/decompression, list intersection, and simple scoring. With the advancement of web search engines, relevance researchers are using more complicated scoring models for ranking documents. The paper mentioned that it is doing the selection only "Thus, our approach can be seen as implementing the first phase, which aims to select promising candidates that the complete scoring function should be applied to. In contrast, the second phase has a very different structure and implementing it on a GPU would be an interesting and challenging problem for future work." Our work is different in that we target the first-stage fast ranking models from the "Selection Phase" as well as the bigger ranking models from second-stage expensive ranking phase "Ranking Phase".

Second, data copy latency is not calculated; they mentioned that it is happening async while the GPU is busy doing other work, but even though it is async, it will affect the query latency. We analyze the ranking with and without data copy. This is beneficial in case of the on-chip GPU, like AMD APUs.

Yan et al. [45] studied using FPGAs in web search engine. While the estimated number of served documents "D" by the query per second "Q" is good, DQ/Dollar = 9.75 compared to baseline 1.36. Moving a big part to the FPGAs from the general purpose CPU architecture is a big investment in new hardware, development, debugging and testing. Also the ability to maintain and change this code on a regular basis is questionable. On the other hand, with our proposal of using the GPU we introduce a change in the code base from C++ to C++ AMP (not a big change). Furthermore, it costs less per machine, around 300-400Dollars cost of the GPU instead of the claimed $4,410.

Putnam et al. [33] the paper uses FPGA to accelerate Microsoft Bing.com. While the paper reported 2× throughput with 50% less power consumption, this work affected the code development time and the code maintainability. Also, many optimization are made to change floating-point operations to fixed point to round to integers. They did not apply the same optimization to the CPU for proper compare. Also, this work uses 7-9 FPGAs in a chain of 7-9 servers connected to perform the ranking task (Because the logic of the ranking can't fit in one FPGA). This complicates the deployment and the fault tolerance story.

Recently FPGA vendors started to support OpenCL. Rodriguez-Donate et al. [36] shows the early experiences with OpenCL on FPGAs. While the optimized code performed well, it suffered from area inefficiency.

Some recent work on the GPUs in the data center by Hetherington et al. [38] shows that data center workload, e.g., Memcached could be 27% more cost efficient than the CPU.

Chapter 7

Conclusion

In this thesis, we examined the use of GPU in the data center to accelerate a regular data center workload, a web search engine. Studying the web search engine's query processing stack in depth allowed us to find the candidate modules for acceleration. Our experimentations reveal the potential to use GPU to achieve vertical scaling with a limited power budget increase. We showed that the GPU can be lower latency than the CPU when using complex rankers and when it is used to rank a large number of documents per query. The reduction of the ranking latency and the increased throughput can be used in four scenarios:

- Adding a commodity GPU to each ISN in the current system without an increase in the index size allows for processing more documents in each ISN per query. This counters the negative effect of the early termination that happens in some cases by allowing more documents to be ranked within the current SLA. The documents are sorted in the index by importance. So including more documents in the selection phase doesn't always mean a better result. But in some rare cases the later documents (document at the end of the index) may be more relevant to the to the user search query.

- Adding GPU per ISN allows a more sophisticated and complex ranking models types to be feasible for relevance researchers to use in the selection stage. This is a great tool for relevance researchers to be able to run more complex models on the selection phase instead of using the fast first-pass rankers. Currently, they are restricted on the size and the complexity of the ranking model that they can use in the selection phase. This restriction is because the ranking on that phase runs on a huge number of documents, and a small increase of ranking latency will accumulate for each document. If that happen, the number of considered documents will be decreased significantly. Our experimentation shows that ranking on GPU can be 19× faster than on the CPU, which allows new types of these rankers to be used on fast first-pass ranking phase. With deeper layers of machine learning

models comes better ranking that leads to a stronger ranking in fast first-pass ranking phase. This allows for a better-reduced set of documents to the second-stage expensive ranking phase that will increase the quality of the documents and their relevance.

- Allows the ISNs to serve a larger index partition. This is because of the latency decrease using GPUs that allow the ISNs to consider more documents from the index.
- Reduce the number of machines in the data center per one full index. This can be achieved by repartitioning the index to bigger chunks in the current system. Assuming that the I/O and caching are not the bottlenecks.

All the previous points can allow for faster ranking on a larger index with cost efficient per query that leads to the increase of the search quality.

7.1 Future Work

GPUs can be more efficient than the CPU for handling the high parallel data driven applications. The following list contains more areas that we would like to examine the possibility of accelerating using the GPUs

- Support more types of ranking models and more feature extraction.
- Examine the benefit of offloading more compute intensive parts of the query processing, e.g. the index decompression. Although this seems like a good candidate, we can't confirm that this is a compute-bound process because it involves I/O too.
- Consider accelerating parts of the top level stack of the query processing, for example, the aggregator that aggregate the scores from all the ISNs.
- Another interesting area is offloading parts of the index building steps to the GPU. Especially with the recent work on using GPUs for the network packets. It could be interesting to examine building crawler on the GPU.

Bibliography

[1] Adnan Abid, Naveed Hussain, Kamran Abid, Farooq Ahmad, Muhammad Shoaib Farooq, Uzma Farooq, Sher Afzal Khan, Yaser Daanial Khan, Muhammad Azhar Naeem, and Nabeel Sabir. A survey on search results diversification techniques. *Neural Computing and Applications*, pages 1–23.

[2] AMD Corp. AMD Accelerated Processing Units. http://fusion.amd.com/, 2012.

[3] AnandTech. Maxwell: Designed For Energy Efficiency. http://www.anandtech.com/show/7764/the-nvidia-geforce-gtx-750-ti-and-gtx-750-review-maxwell/3, 2012.

[4] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. ACM Conf. on World Wide Web*, pages 107–117, 1998.

[5] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International journal of human-computer interaction*, 17(3):333–356, 2004.

[6] Maurice De Kunder. The size of the world wide web. *WorldWideWebSize*, 2012.

[7] Electronic Educational Devices. Watts up pro, 2009.

[8] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using Graphics Processors for High Performance IR Query Processing. In *Proc. ACM Conf. on World Wide Web*, pages 421–430, 2009.

[9] Dennis F Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5(1):1, 2004.

[10] Boby George and Pooja Nagpal. Optimizing parallel applications using concurrency visualizer: A case study. *Parallel Computing Platform Group, Microsoft Corporation*, 2010.

[11] Kate Gregory and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, 2012.

[12] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[13] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[14] Facts Hunt. Total number of websites & size of the internet as of 2013, 2015.

[15] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pages 314–325, 2010.

[16] Thienne Johnson, Carlos Acedo, SG Kobourov, and Sabrina Nusrat. Analyzing the evolution of the internet. In *17th IEEE Eurographics Conference on Visualization (EuroVis–short papers). Accepted, to appear in*, 2015.

[17] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, page 9, 2011.

[18] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[19] 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 *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pages 451–460, 2010.

[20] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[21] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[22] Microsoft Corp. C++ AMP Overview. http://msdn.microsoft.com/en-us/library/vstudio/hh265136.aspx.

[23] Microsoft Corp. *Microsoft Windows High-Resolution Performance Timer API*, 2012.

[24] Brian H Murray and Alvin Moore. Sizing the internet. *White paper, Cyveillance*, page 3, 2000.

[25] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. *ACM SIGPLAN Notices*, 49(4):529–542, 2014.

[26] Johan Nilsson. Timers-implement a continuously updating, high-resolution time provider for windows. *MSDN Magazine*, pages 78–88, 2004.

[27] Alexandros Ntoulas, Marc Najork, Mark Manasse, and Dennis Fetterly. Detecting spam web pages through content analysis. In *Proceedings of the 15th international conference on World Wide Web*, pages 83–92. ACM, 2006.

[28] CUDA Nvidia. Programming guide, 2008.

[29] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[30] NVIDIA Corp. NVIDIA PerfKit. https://developer.nvidia.com/nvidia-perfkit, 2013.

[31] Kyoung-Su Oh and Keechul Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[33] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A reconfigurable fabric

for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[34] Feng Qiu, Zhenyu Liu, and Junghoo Cho. Analysis of user web traffic with a focus on search activities. In *WebDB*, pages 103–108. Citeseer, 2005.

[35] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Proc. Text Retrieval Conference (TREC)*, pages 109–126, 1994.

[36] C Rodriguez-Donate, G Botella, C Garcia, E Cabal-Yepez, and M Prieto-Matias. Early experiences with opencl on fpgas: Convolution case study. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 235–235. IEEE, 2015.

[37] Dillon Sharlet, Aaron Kunze, Stephen Junkins, and Deepti Joshi. Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL. In *General Meeting of LLVM Developers and Users*, 2012.

[38] Tor M. Aamodt Tayler H. Hetherington, Mike O'Connor. Memcachedgpu: Scaling-up scale-out key-value stores. 2015.

[39] TechPowerUp. TechPowerUp GPU-Z. http://www.techpowerup.com/gpuz/, 2012.

[40] WorldOMeters. Internet live stats. http://www.internetlivestats.com/watch/internet-users/.

[41] Yeming Xiao. Speeding up deep neural network based speech recognition systems. *Journal of Software*, 9(10):2706–2712, 2014.

[42] Anhao Xing, Xin Jin, Ta Li, Xuyang Wang, Jielin Pan, and Yonghong Yan. Speeding up deep neural networks for speech recognition on arm cortex-a series processors. In *Natural Computation (ICNC), 2014 10th International Conference on*, pages 123–127. IEEE, 2014.

[43] Song Xing and Bernd-Peter Paris. Measuring the size of the internet via importance sampling. *Selected Areas in Communications, IEEE Journal on*, 21(6):922–933, 2003.

[44] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web*, pages 401–410. ACM, 2009.

[45] Jing Yan, Zhan-Xiang Zhao, Ning-Yi Xu, Xi Jin, Lin-Tao Zhang, and Feng-Hsiung Hsu. Efficient query processing for web search engine with fpgas. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 97–100, 2012.

[46] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.

[47] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for fsm parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 619–630. ACM, 2015.

[48] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 543–558. ACM, 2014.

[49] Justin Zobel and Alistair Moffat. Inverted Files for Text Search Engines. *ACM Comput. Surv.*, 38(2), July 2006.